US006243827B1

(12) **United States Patent**
Renner, Jr.

(10) **Patent No.: US 6,243,827 B1**
(45) **Date of Patent: Jun. 5, 2001**

(54) **MULTIPLE-CHANNEL FAILURE DETECTION IN RAID SYSTEMS**

(75) Inventor: **William F. Renner, Jr.**, Baltimore, MD (US)

(73) Assignee: **Digi-Data Corporation**, Jessup, MD (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/108,015**

(22) Filed: **Jun. 30, 1998**

(51) **Int. Cl.**[7] **H05K 10/00**
(52) **U.S. Cl.** **714/6**
(58) **Field of Search** 711/112, 114; 714/2, 6

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,598,357	7/1986	Swenson et al. .	
4,945,535	7/1990	Hosotani et al. .	
5,166,936	11/1992	Ewert et al. .	
5,249,288	9/1993	Ippolito et al. .	
5,271,012	12/1993	Blaum et al. .	
5,274,799	12/1993	Brant et al. .	
5,285,451	2/1994	Henson et al. .	
5,412,661	5/1995	Hao et al. .	
5,418,921 *	5/1995	Cortney et al.	711/114
5,463,765	10/1995	Kakuta et al. .	
5,469,453	11/1995	Glider et al. .	
5,479,611	12/1995	Oyama .	
5,526,482	6/1996	Stallmo et al. .	
5,548,711	8/1996	Brant et al. .	
5,564,011	10/1996	Yammine et al. .	
5,572,659	11/1996	Iwasa et al. .	
5,574,856	11/1996	Morgan et al. .	
5,574,882	11/1996	Menon et al. .	
5,600,783	2/1997	Kakuta et al. .	
5,617,425	4/1997	Anderson .	
5,636,359	6/1997	Beardsley et al. .	
5,644,697	7/1997	Matsumoto et al. .	
5,657,439	8/1997	Jones et al. .	
5,889,934 *	3/1999	Peterson	714/6
5,974,544 *	10/1999	Jeffries et al.	713/1

OTHER PUBLICATIONS

Ridge, Peter M. The Book of SCSI: A Guide for Adventurers. Chapter H: An Introduction to RAID. pp. 323–329. 1995 William Pollock, publisher.

Pankaj Jalote Fault Tolerance in Distributed Systems. Section 3.3.1 Problem Definition, pp. 100–101. 1994 P T R Prentice Hall.

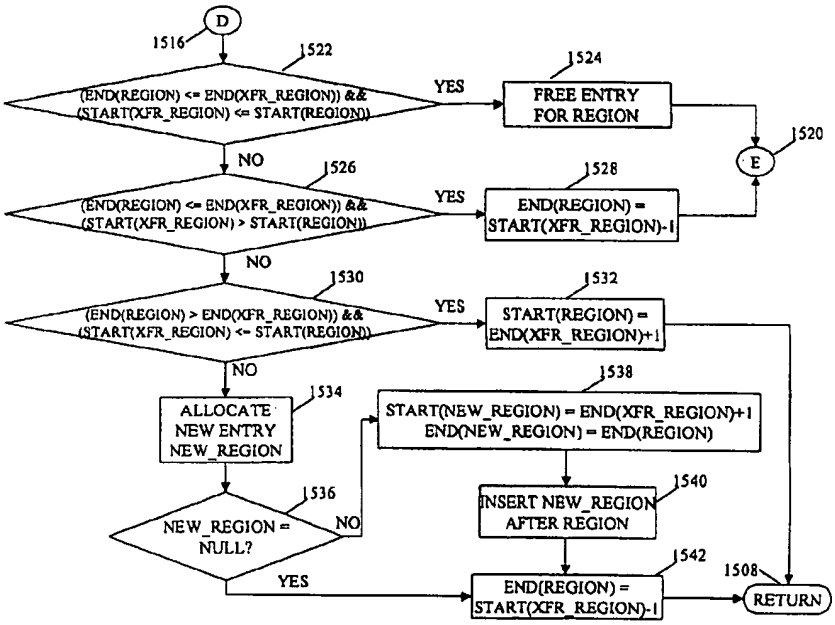
* cited by examiner

Primary Examiner—John F. Niebling
Assistant Examiner—Stacy A Whitmore
(74) *Attorney, Agent, or Firm*—William S. Ramsey

(57) **ABSTRACT**

This invention is a software-based method for facilitating the recovery of a RAID storage system from the simultaneous failure of two or more disks (catastrophic failure). It involves the identification of the logical address and length of the failed areas of the failed disks and the writing of this information into a bad region table which is replicated on each disk. This makes it easier and less expensive to identify the problem areas and make the necessary repairs.

4 Claims, 17 Drawing Sheets



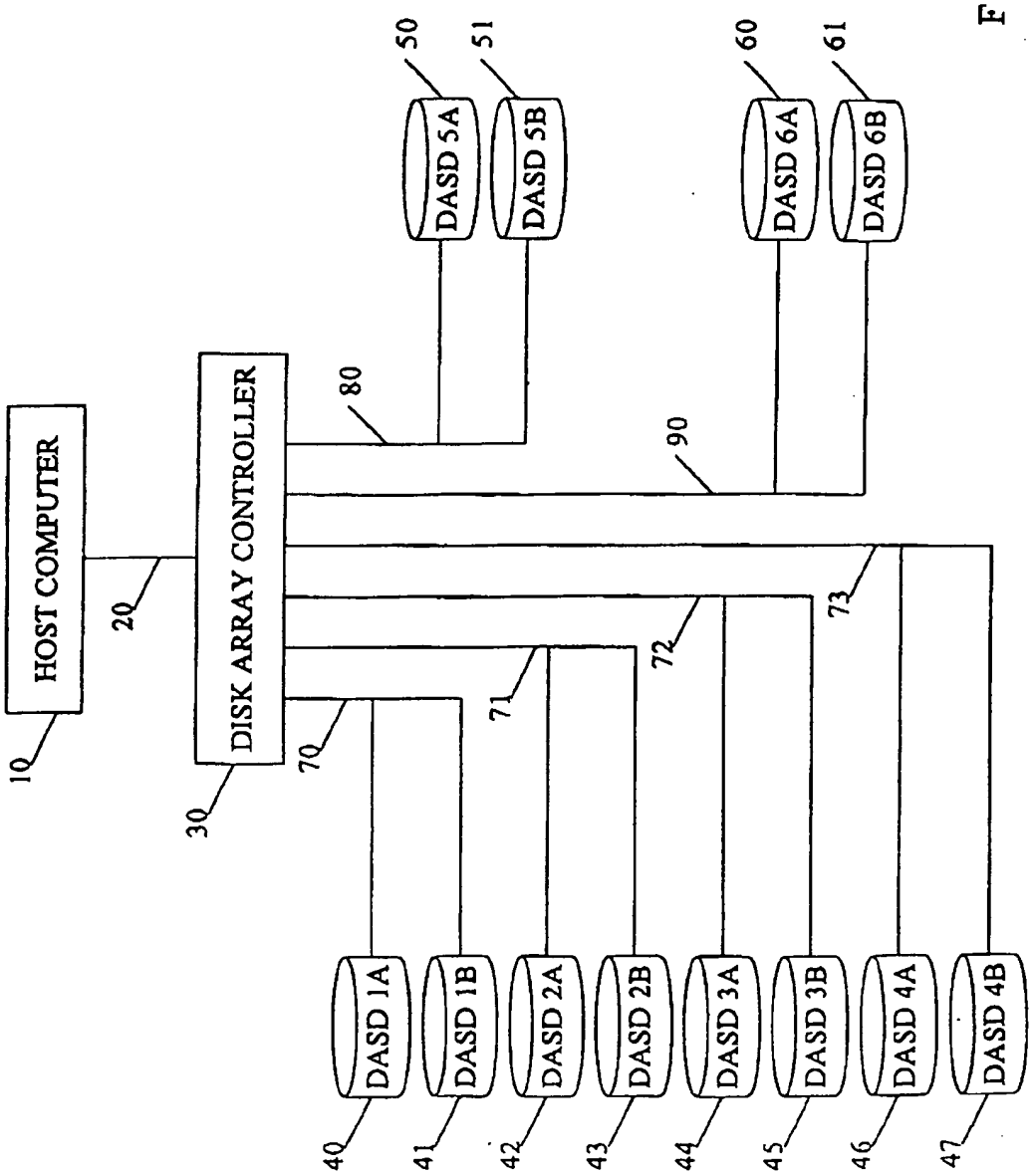

HOST COMPUTER
10
20
DISK ARRAY CONTROLLER
30
80
90
70
71
72
73
50
51
60
61
DASD 5A
DASD 5B
DASD 6A
DASD 6B
DASD 1A
DASD 1B
DASD 2A
DASD 2B
DASD 3A
DASD 3B
DASD 4A
DASD 4B
40
41
42
43
44
45
46
47

Figure 1

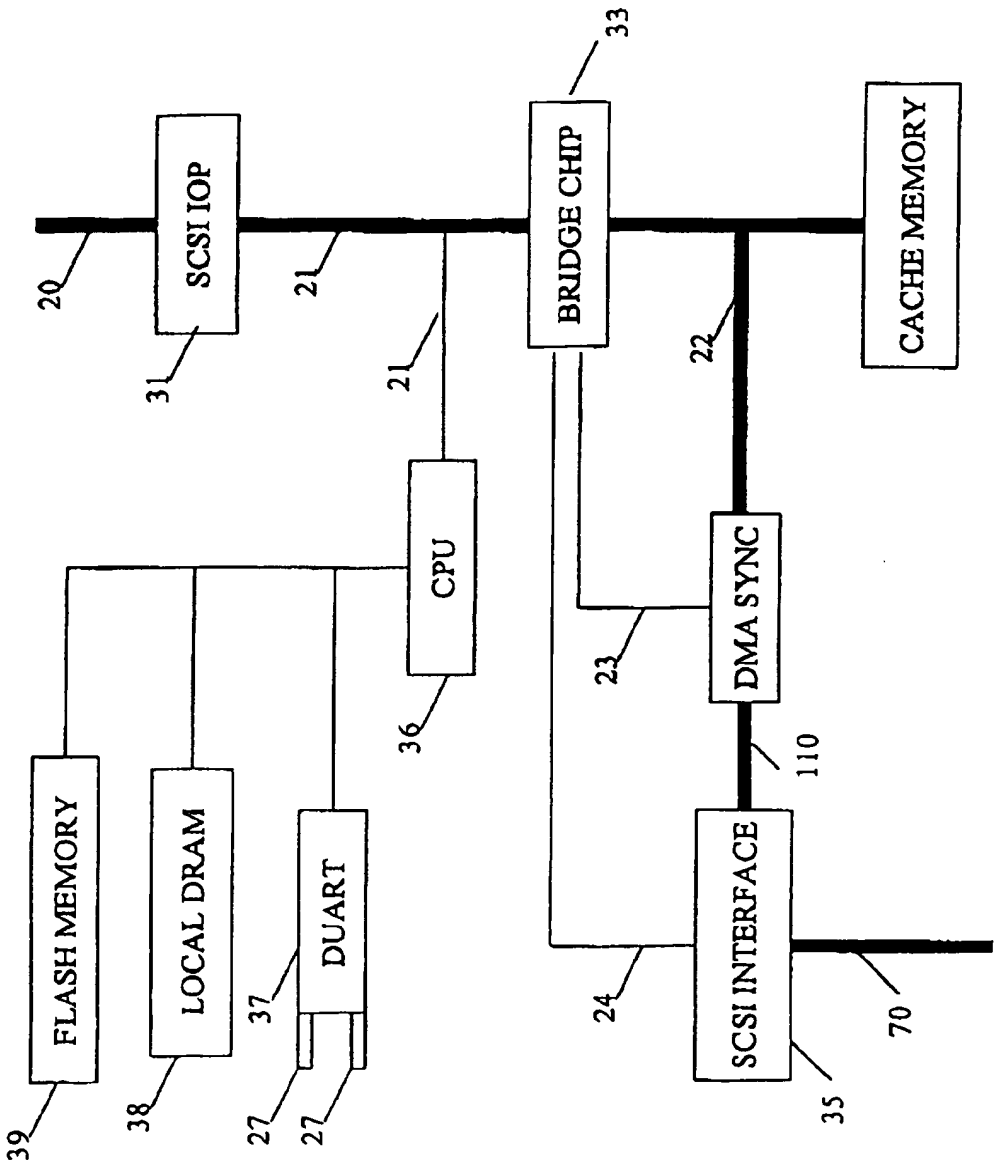

SCSI IOP
BRIDGE CHIP
CACHE MEMORY
CPU
DMA SYNC
SCSI INTERFACE
FLASH MEMORY
LOCAL DRAM
DUART
20
21
21
22
23
24
27
27
31
33
35
36
37
38
39
70
110

Figure 2

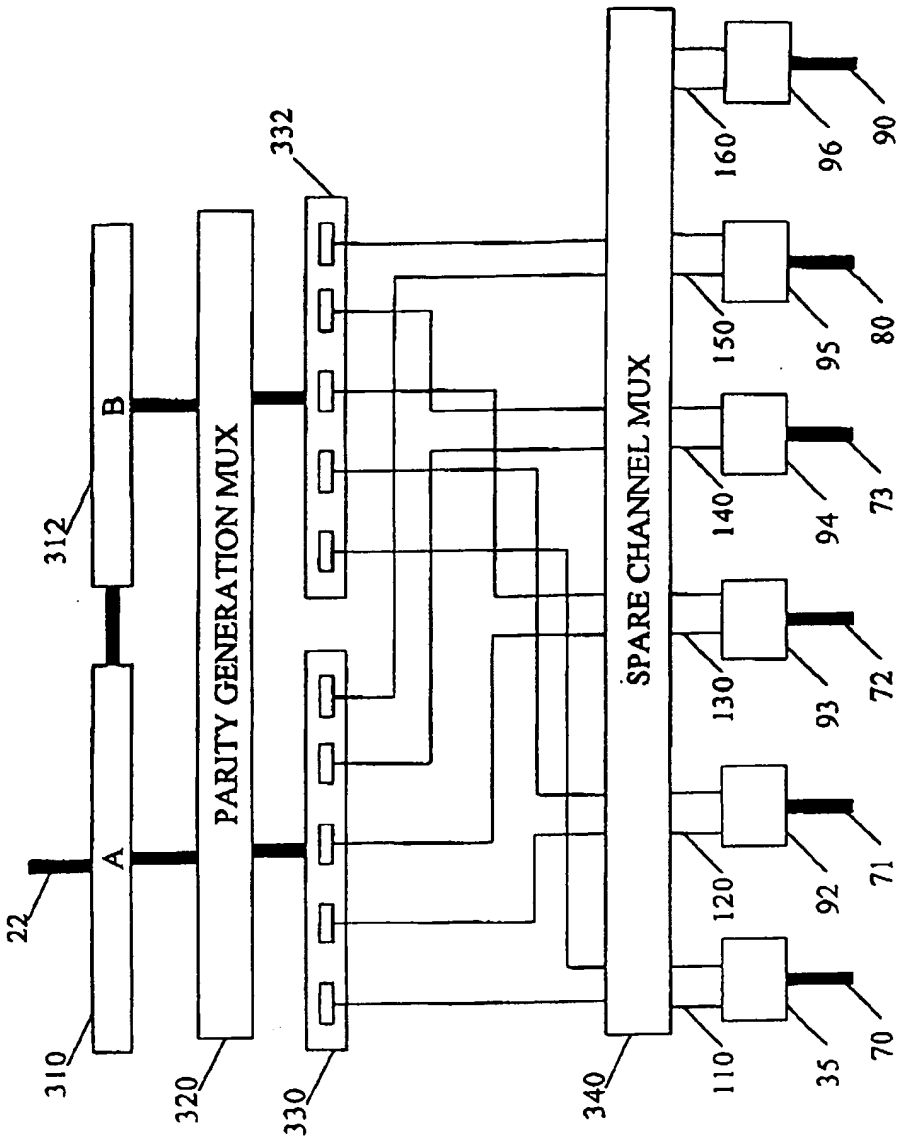

22
310
A
312
B
320
PARITY GENERATION MUX
330
332
340
SPARE CHANNEL MUX
110
120
130
140
150
160
35
92
93
94
95
96
70
71
72
73
80
90

Figure 3

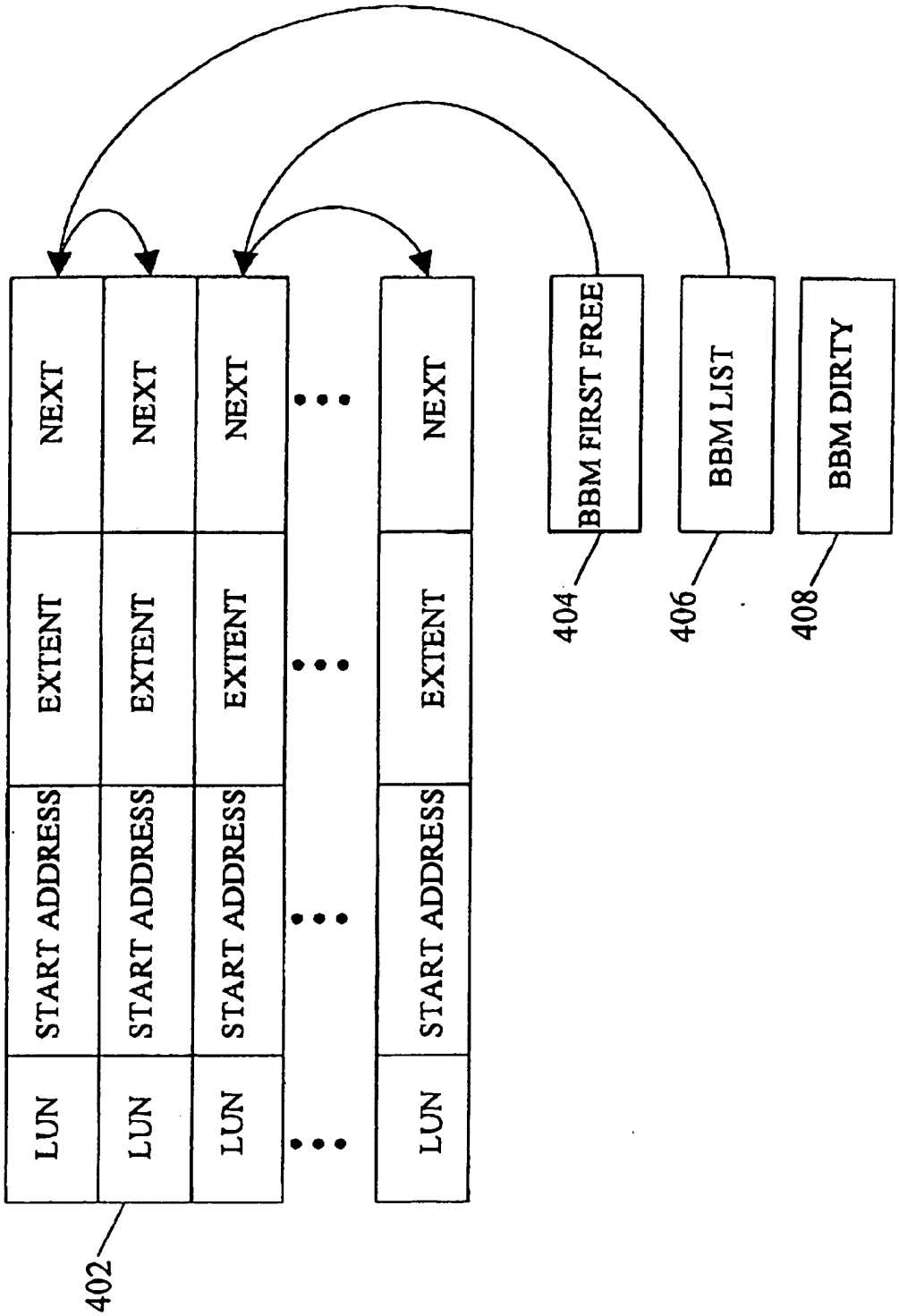

NEXT
NEXT
NEXT
NEXT
EXTENT
EXTENT
EXTENT
EXTENT
START ADDRESS
START ADDRESS
START ADDRESS
START ADDRESS
LUN
LUN
LUN
LUN
402
BBM FIRST FREE
BBM LIST
BBM DIRTY
404
406
408

Figure 4

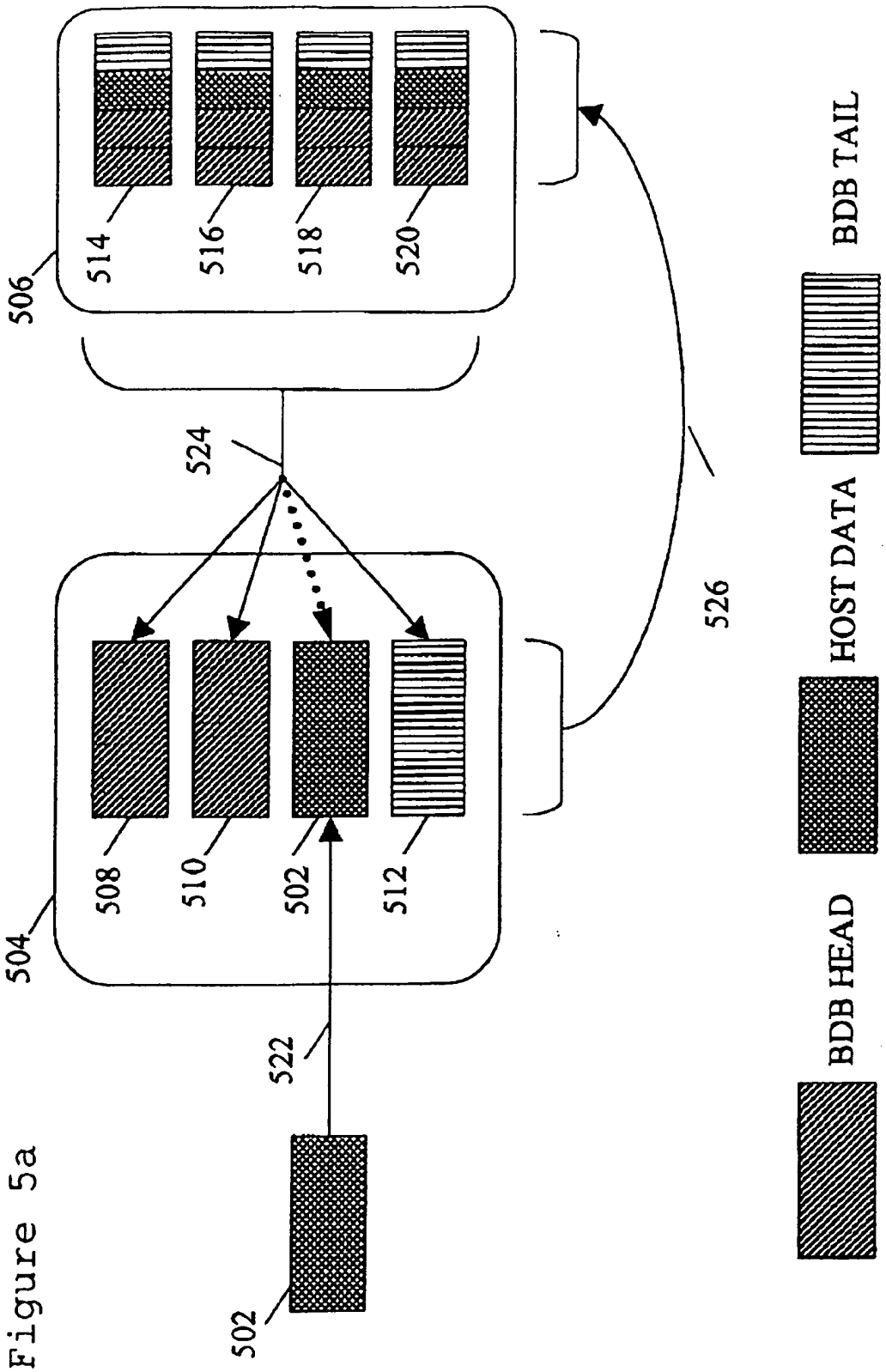

506
514
516
518
520
524
526
504
508
510
502
512
522
502
BDB TAIL
HOST DATA
BDB HEAD

Figure 5a

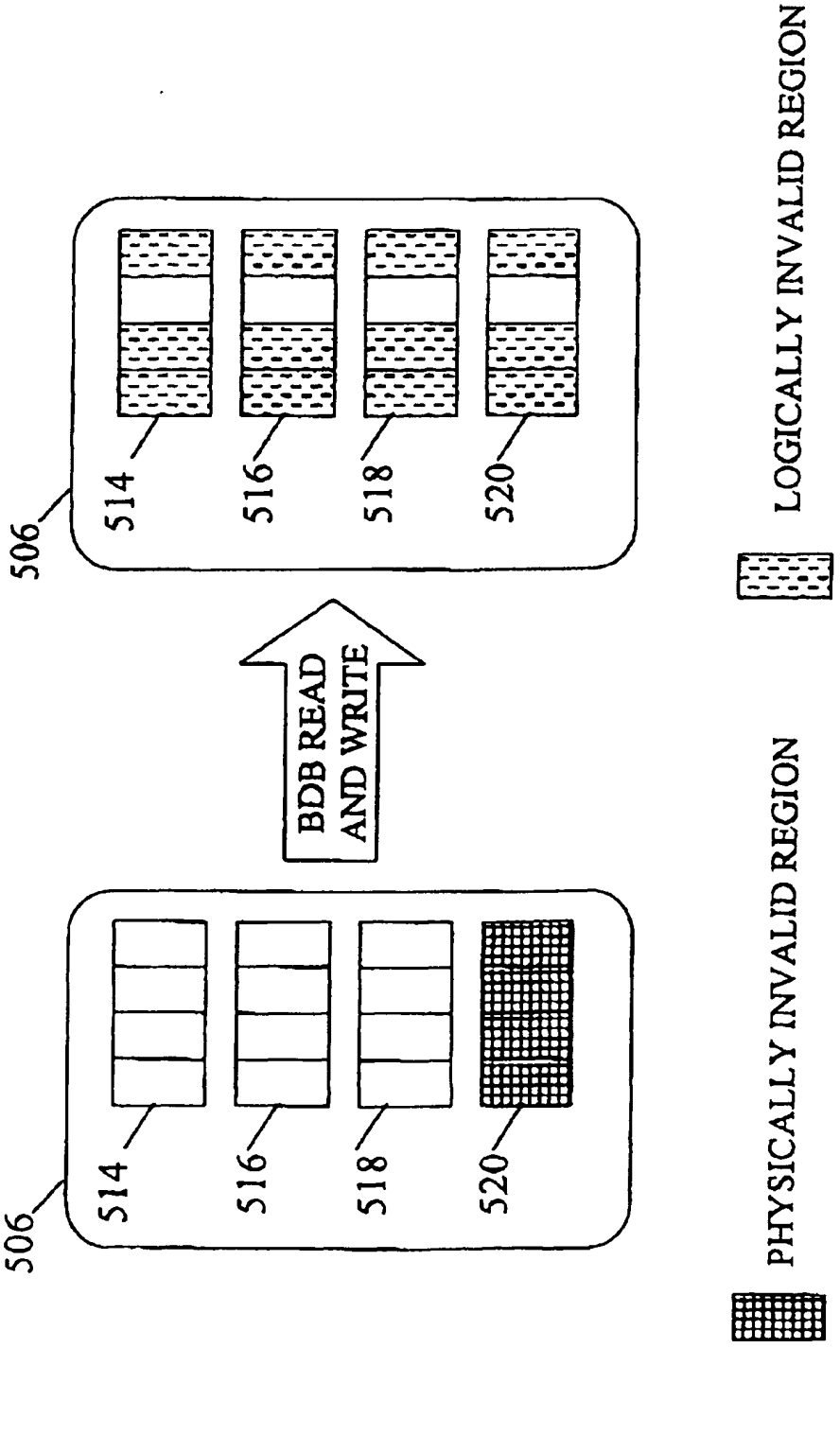
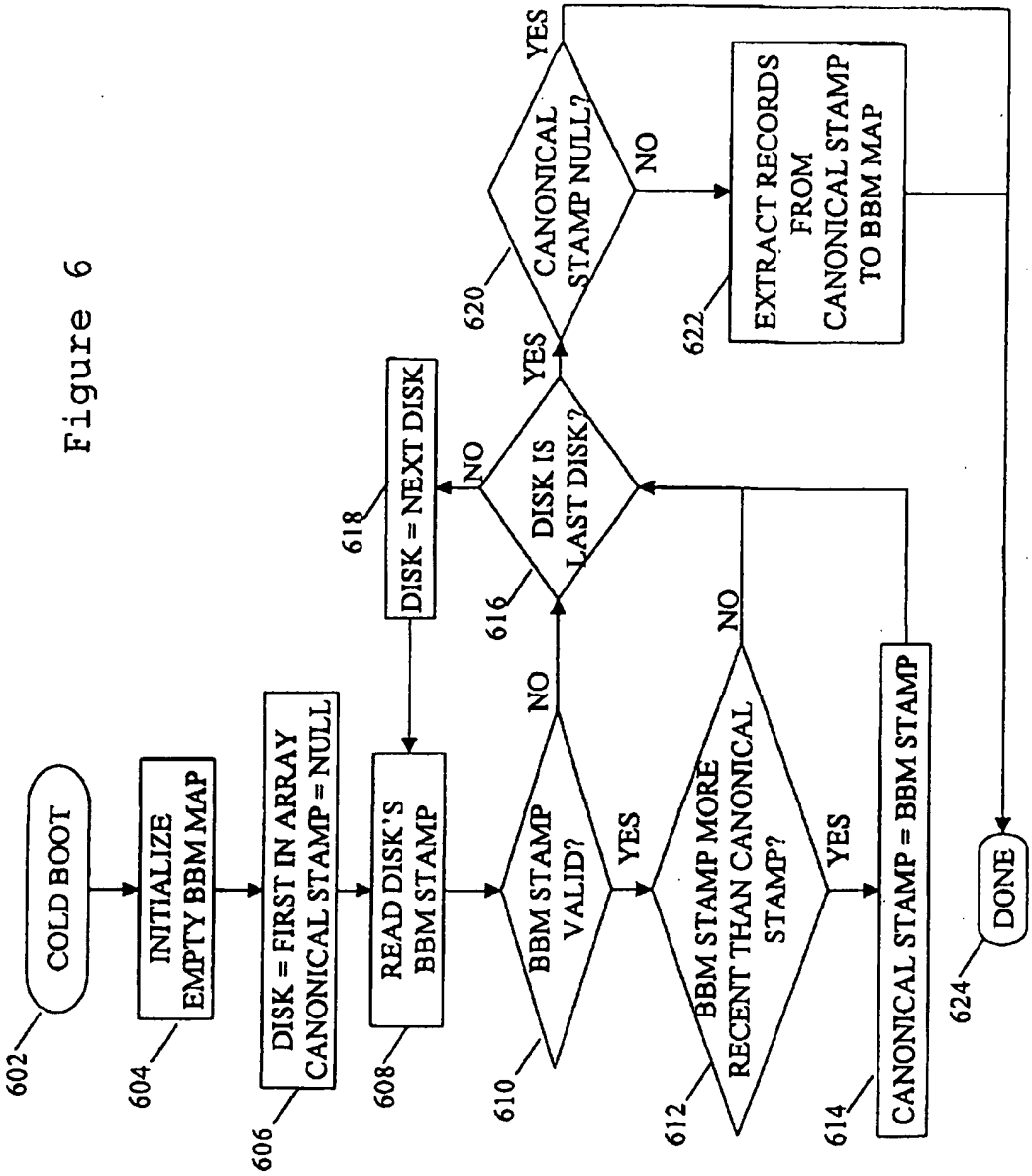

506
514
516
518
520
BDB READ AND WRITE
506
514
516
518
520
LOGICALLY INVALID REGION
PHYSICALLY INVALID REGION

Figure 5b

Figure 6


COLD BOOT
602
INITIALIZE EMPTY BBM MAP
604
DISK = FIRST IN ARRAY CANONICAL STAMP = NULL
606
READ DISK'S BBM STAMP
608
BBM STAMP VALID?
610
NO
YES
BBM STAMP MORE RECENT THAN CANONICAL STAMP?
612
NO
YES
CANONICAL STAMP = BBM STAMP
614
DISK IS LAST DISK?
616
NO
YES
DISK = NEXT DISK
618
CANONICAL STAMP NULL?
620
YES
NO
EXTRACT RECORDS FROM CANONICAL STAMP TO BBM MAP
622
DONE
624

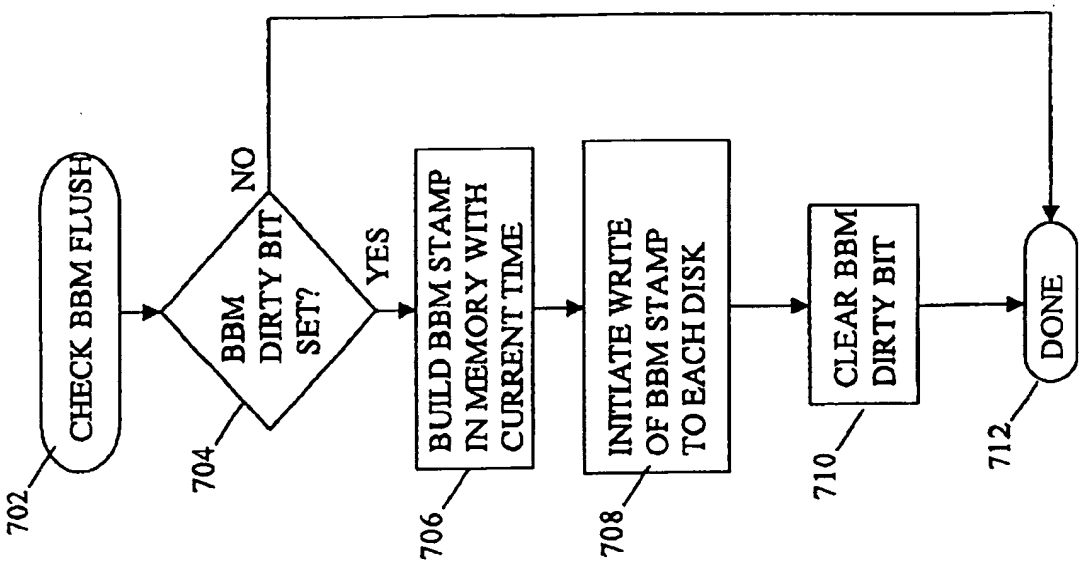

CHECK BBM FLUSH
BBM DIRTY BIT SET?
NO
YES
BUILD BBM STAMP IN MEMORY WITH CURRENT TIME
INITIATE WRITE OF BBM STAMP TO EACH DISK
CLEAR BBM DIRTY BIT
DONE
702
704
706
708
710
712

Figure 7

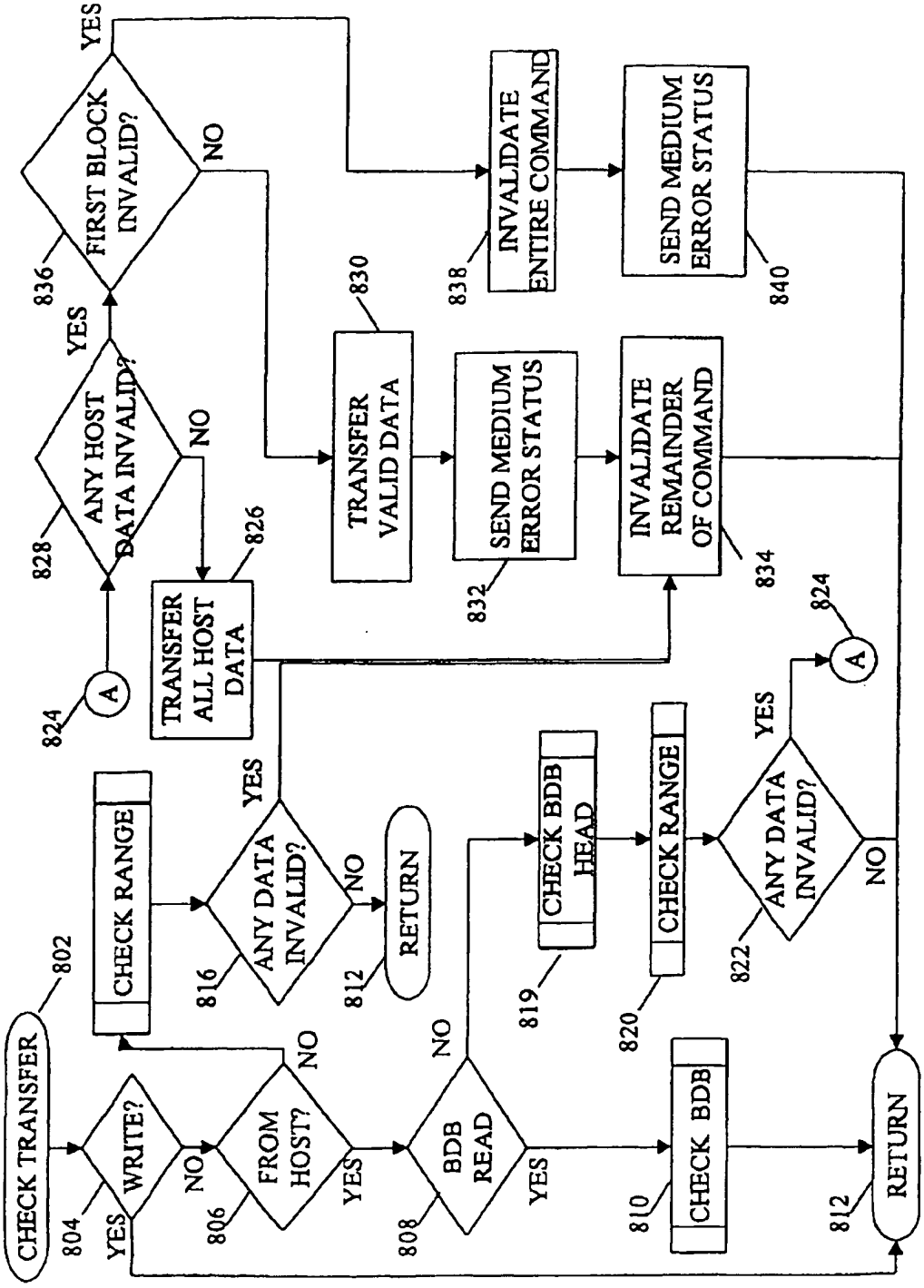

CHECK TRANSFER
802
WRITE?
804
YES
NO
FROM HOST?
806
YES
NO
BDB READ
808
YES
NO
CHECK BDB
810
RETURN
812
CHECK RANGE
ANY DATA INVALID?
816
YES
NO
RETURN
812
CHECK BDB HEAD
819
CHECK RANGE
820
ANY DATA INVALID?
822
YES
NO
A
824
A
824
ANY HOST DATA INVALID?
828
YES
NO
FIRST BLOCK INVALID?
836
YES
NO
TRANSFER ALL HOST DATA
826
TRANSFER VALID DATA
830
SEND MEDIUM ERROR STATUS
832
INVALIDATE REMAINDER OF COMMAND
834
INVALIDATE ENTIRE COMMAND
838
SEND MEDIUM ERROR STATUS
840

Figure 8

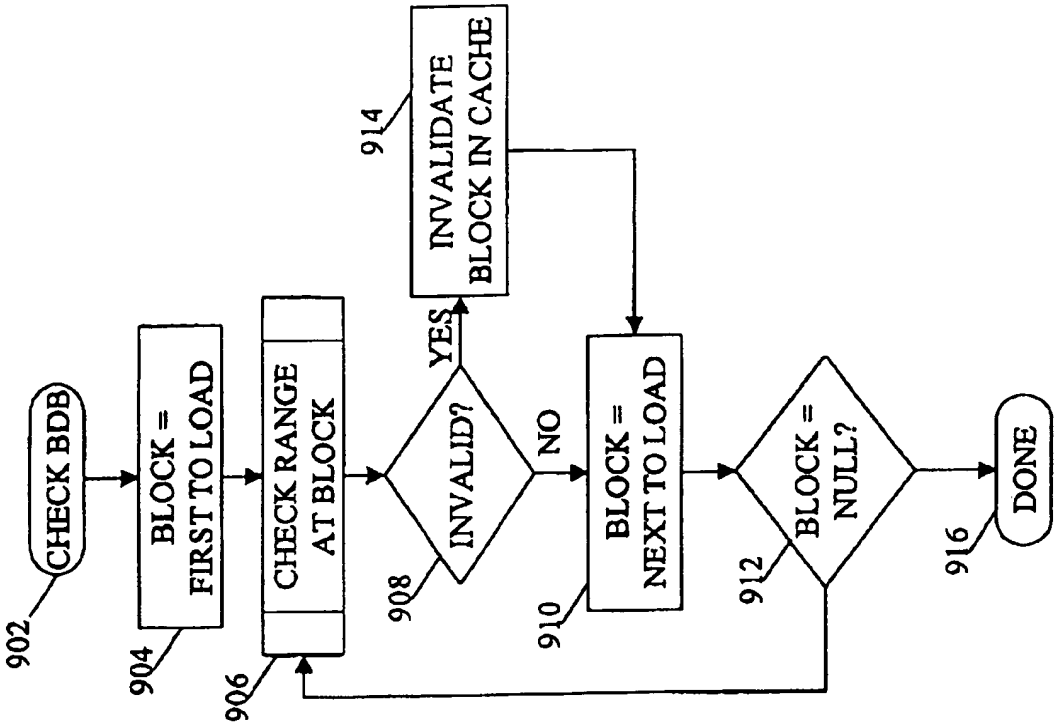

CHECK BDB
902
BLOCK = FIRST TO LOAD
904
CHECK RANGE AT BLOCK
906
INVALID?
908
YES
INVALIDATE BLOCK IN CACHE
914
NO
BLOCK = NEXT TO LOAD
910
BLOCK = NULL?
912
DONE
916

Figure 9

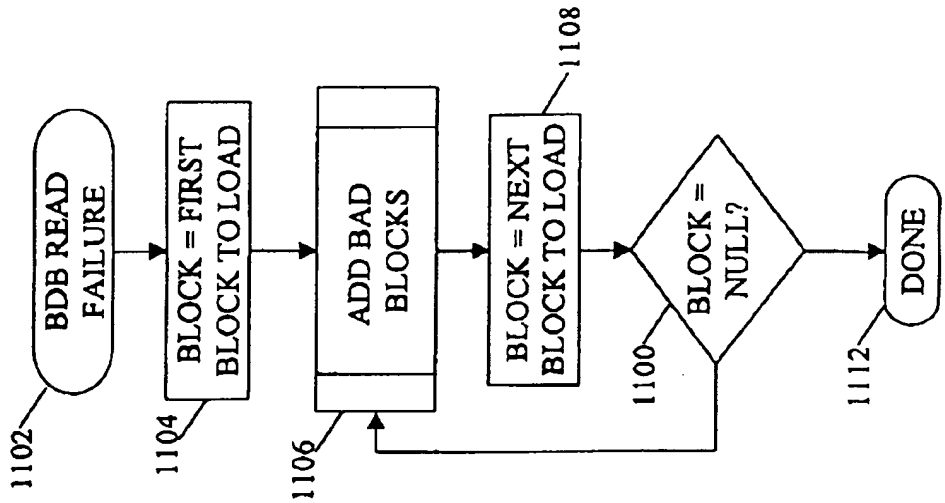


Figure 11

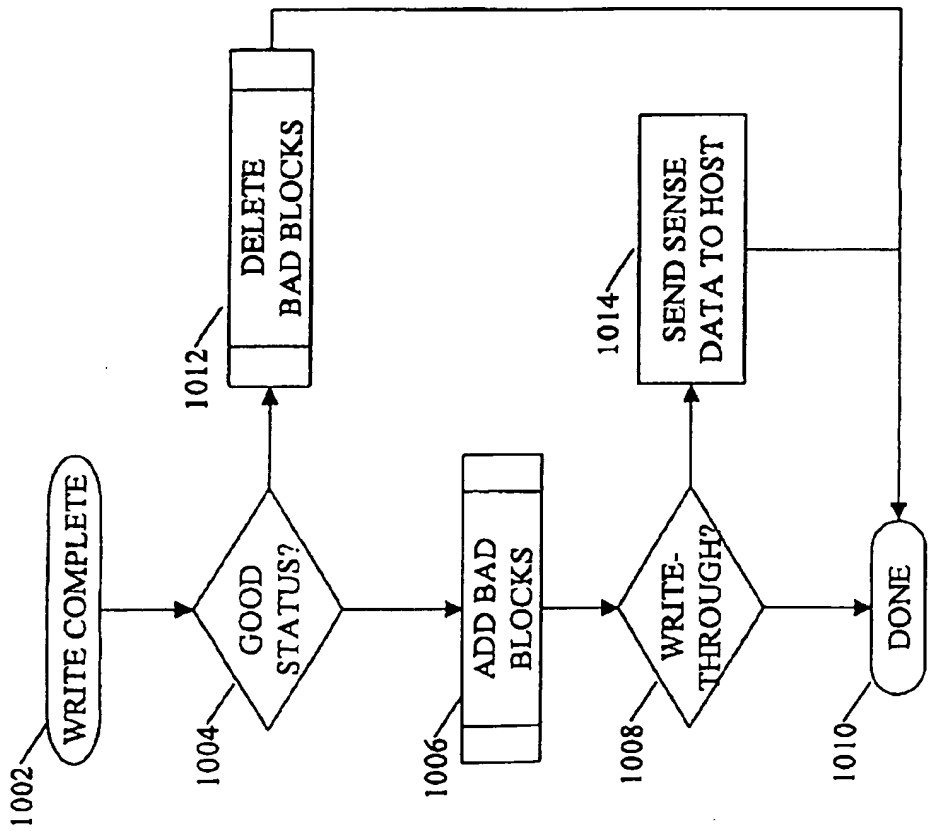


Figure 10

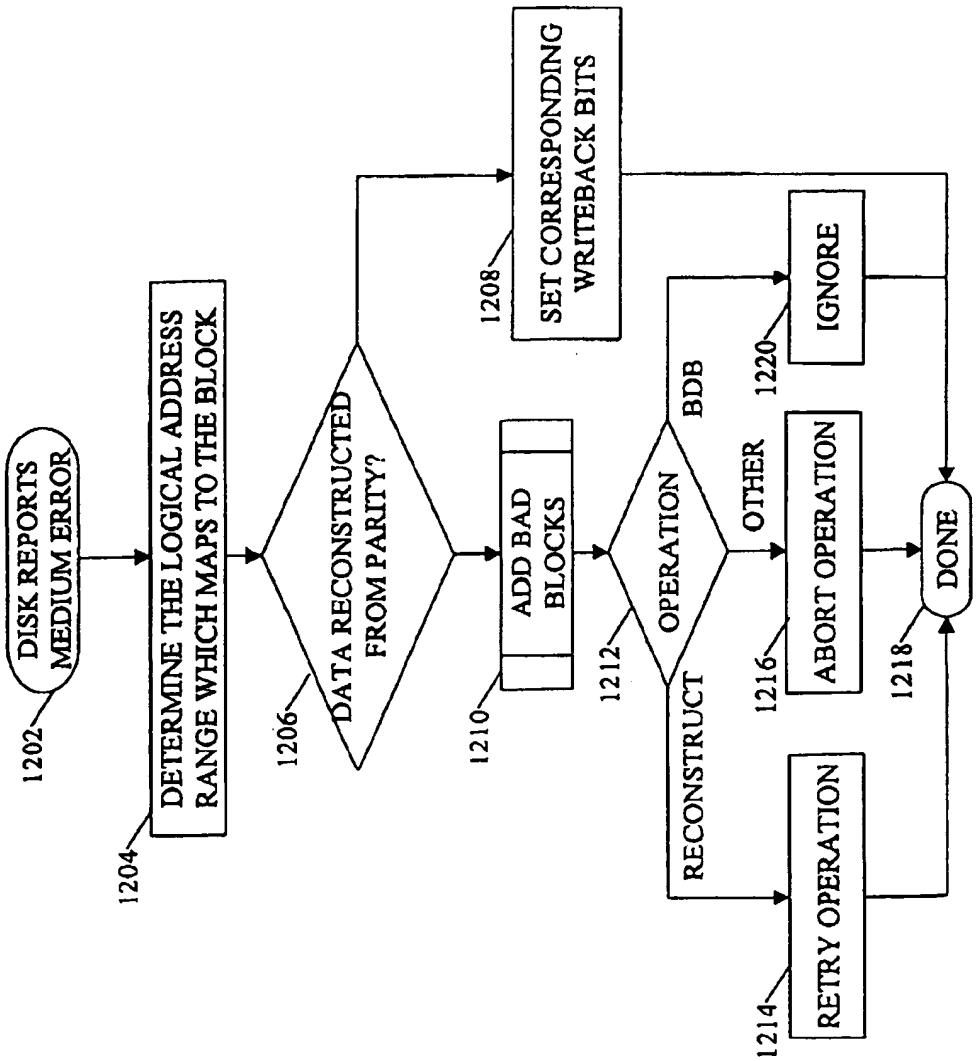

DISK REPORTS MEDIUM ERROR
1202
DETERMINE THE LOGICAL ADDRESS RANGE WHICH MAPS TO THE BLOCK
1204
DATA RECONSTRUCTED FROM PARITY?
1206
SET CORRESPONDING WRITEBACK BITS
1208
ADD BAD BLOCKS
1210
OPERATION
1212
BDB
IGNORE
1220
OTHER
ABORT OPERATION
1216
RECONSTRUCT
RETRY OPERATION
1214
DONE
1218

Figure 12

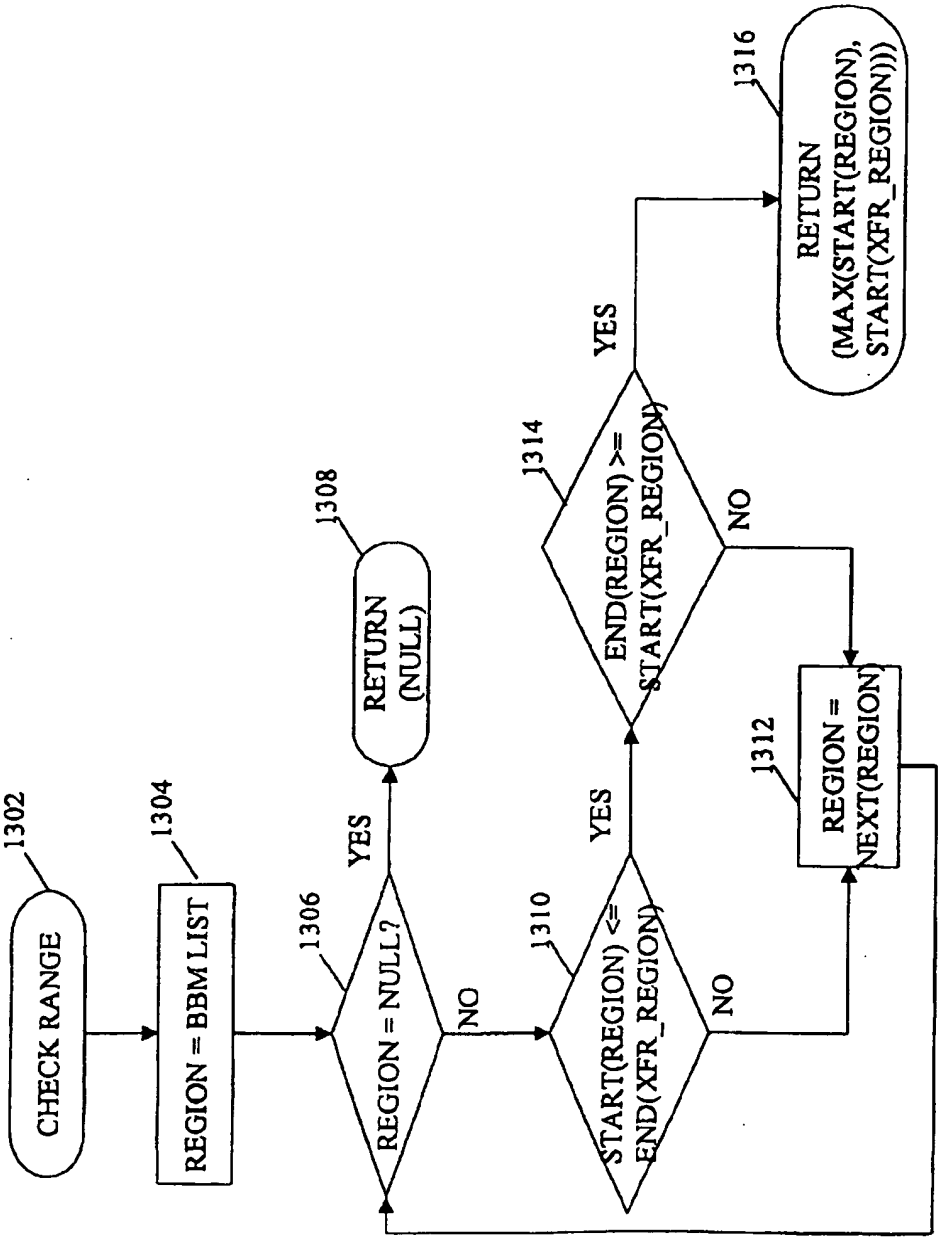

1302
CHECK RANGE
1304
REGION = BBM LIST
1306
REGION = NULL?
YES
NO
1308
RETURN (NULL)
1310
START(REGION) <= END(XFR_REGION)
YES
NO
1312
REGION = NEXT(REGION)
1314
END(REGION) >= START(XFR_REGION)
YES
NO
1316
RETURN (MAX(START(REGION), START(XFR_REGION)))

Figure 13

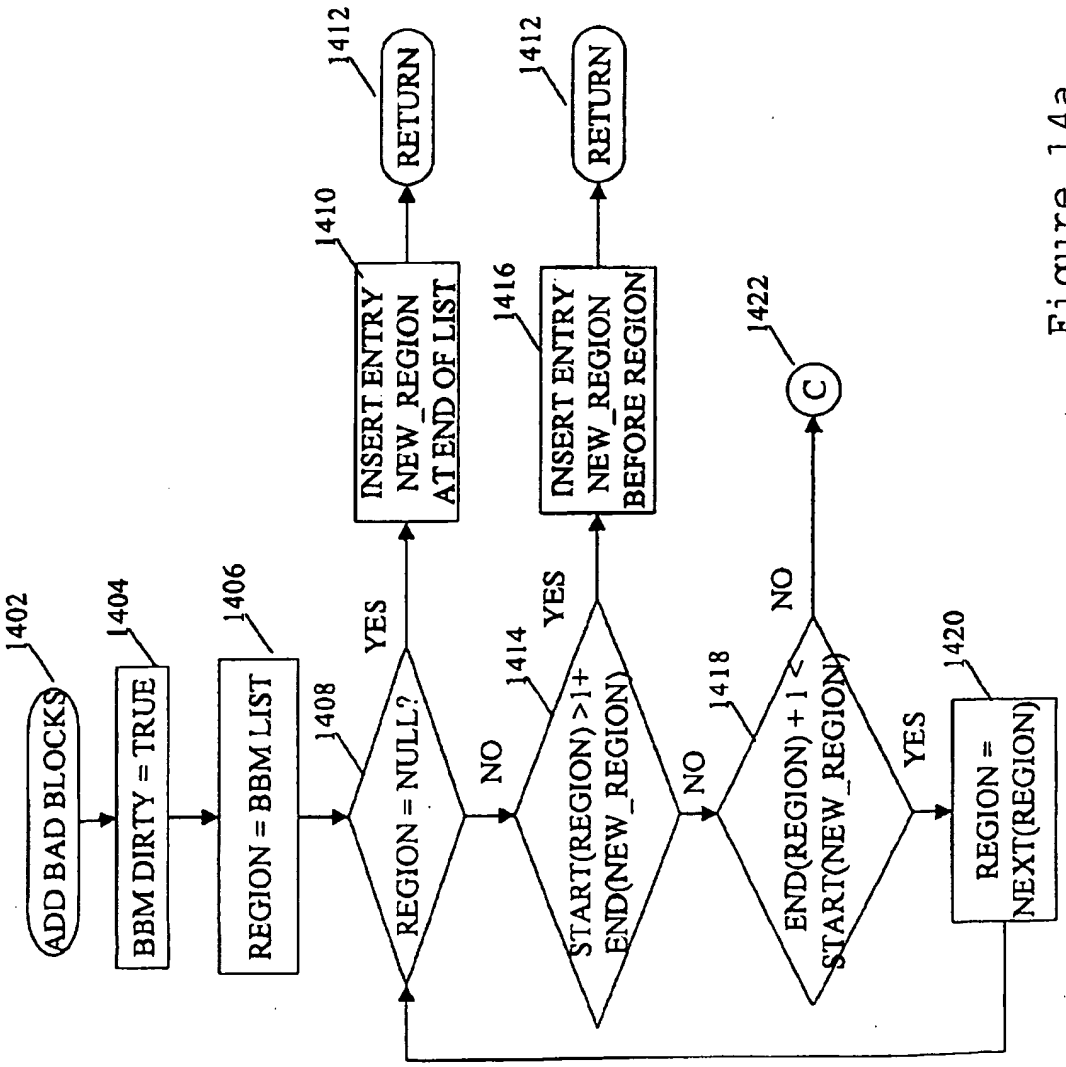

1402
ADD BAD BLOCKS
1404
BBM DIRTY = TRUE
1406
REGION = BBM LIST
1408
REGION = NULL?
YES
NO
1410
INSERT ENTRY NEW_REGION AT END OF LIST
1412
RETURN
1414
START(REGION) >1+ END(NEW_REGION)
YES
NO
1416
INSERT ENTRY NEW_REGION BEFORE REGION
1412
RETURN
1418
END(REGION) + 1 < START(NEW_REGION)
NO
YES
1422
C
1420
REGION = NEXT(REGION)

Figure 14a

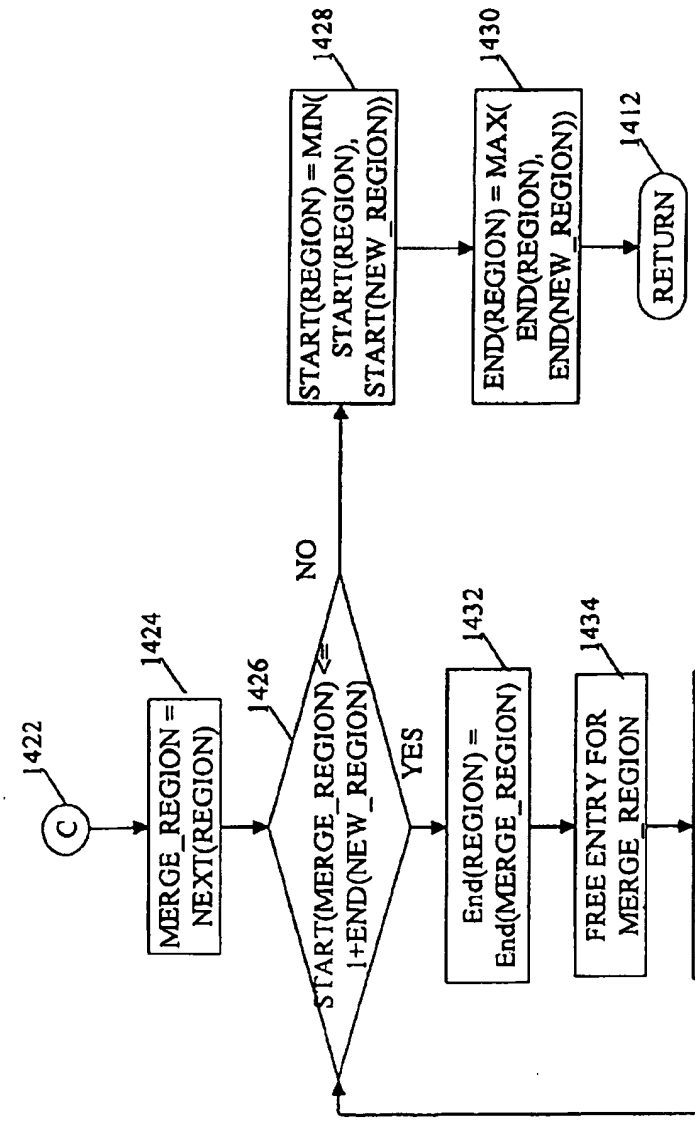

1422
C
1424
MERGE_REGION = NEXT(REGION)
1426
START(MERGE_REGION) <= 1+END(NEW_REGION)
NO
YES
1428
START(REGION) = MIN(START(REGION), START(NEW_REGION))
1430
END(REGION) = MAX(END(REGION), END(NEW_REGION))
1412
RETURN
1432
End(REGION) = End(MERGE_REGION)
1434
FREE ENTRY FOR MERGE_REGION
1436
MERGE_REGION = NEXT(MERGE_REGION)

Figure 14b

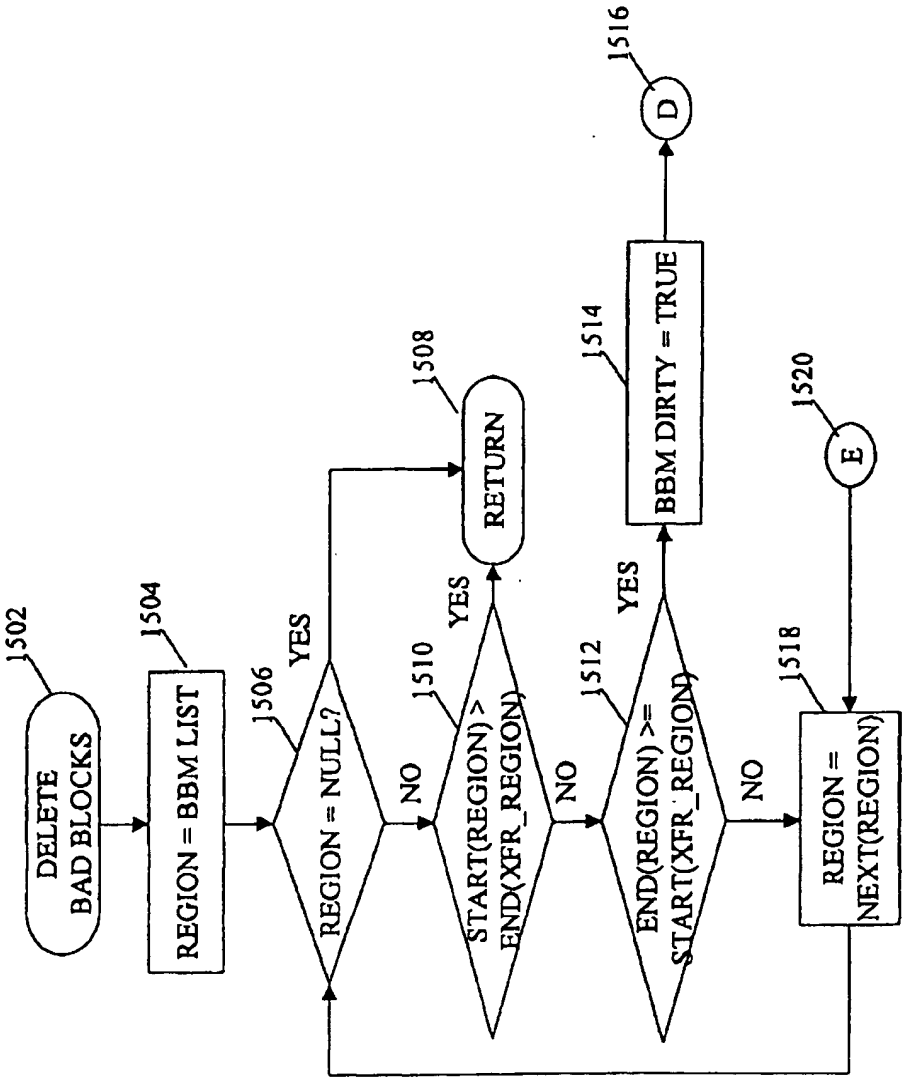

1502
DELETE BAD BLOCKS
1504
REGION = BBM LIST
1506
REGION = NULL?
YES
NO
1508
RETURN
1510
START(REGION) > END(XFR_REGION)
YES
NO
1512
END(REGION) >= START(XFR_REGION)
YES
NO
1514
BBM DIRTY = TRUE
1516
D
1518
REGION = NEXT(REGION)
1520
E

Figure 15a

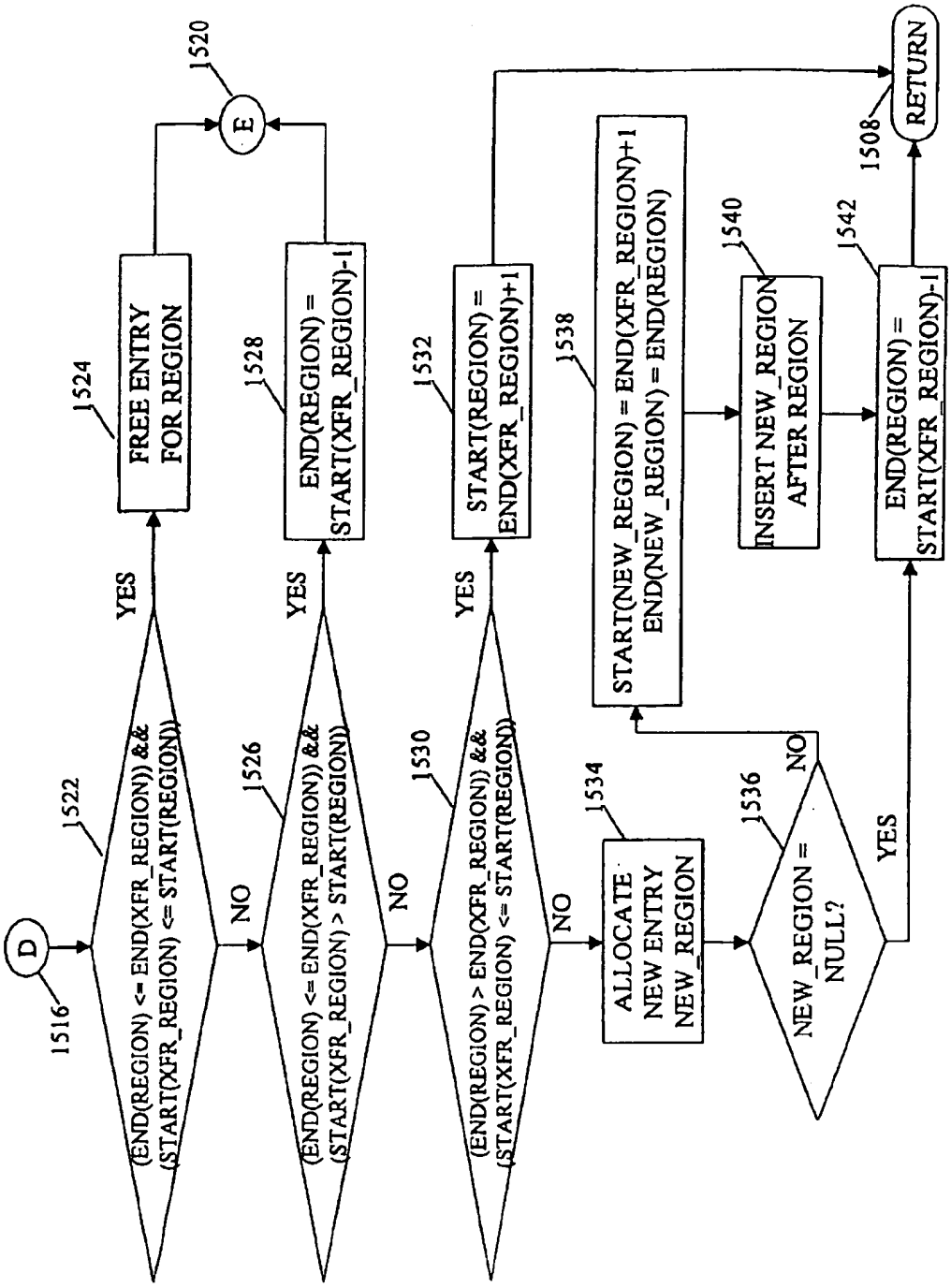

D
1516
1522
(END(REGION) <= END(XFR_REGION)) && (START(XFR_REGION) <= START(REGION))
YES
1524
FREE ENTRY FOR REGION
E
1520
NO
1526
(END(REGION) <= END(XFR_REGION)) && (START(XFR_REGION) > START(REGION))
YES
1528
END(REGION) = START(XFR_REGION)-1
NO
1530
(END(REGION) > END(XFR_REGION)) && (START(XFR_REGION) <= START(REGION))
YES
1532
START(REGION) = END(XFR_REGION)+1
NO
1534
ALLOCATE NEW ENTRY NEW_REGION
1536
NEW_REGION = NULL?
NO
1538
START(NEW_REGION) = END(XFR_REGION)+1 END(NEW_REGION) = END(REGION)
1540
INSERT NEW_REGION AFTER REGION
1542
END(REGION) = START(XFR_REGION)-1
YES
1508
RETURN

Figure 15b

MULTIPLE-CHANNEL FAILURE DETECTION IN RAID SYSTEMS

CROSS-REFERENCE TO RELATED APPLICATIONS.

Not Applicable.

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT.

Not Applicable.

BACKGROUND OF THE INVENTION

(1) FIELD OF THE INVENTION

This invention relates to RAID systems in which multiple-channel failure is detected and the diagnostic information recorded.

(2) DESCRIPTION OF RELATED ART INCLUDING INFORMATION DISCLOSED UNDER 37 CFR 1.97 AND 37 CFR 1.98.

The acronym RAID refers to systems which combine disk drives for the storage of large amounts of data. In RAID systems the data is recorded by dividing each disk into stripes, while the data are interleaved so the combined storage space consists of stripes from each disk. RAID systems fall under 5 different architectures, plus one additional type, RAID-0, which is simply an array of disks and does not offer any fault tolerance. RAID 1–5 systems use various combinations of redundancy, spare disks, and parity analysis to achieve conservation reading and writing of data in the face of one and, in some cases, multiple intermediate or permanent disk failures. Ridge, P. M. *The Book Of SCCSI: A Guide For Adventurers*. Daly City Cal. No Starch Press. 1995 p. 323–329.

It is important to note that multiple disk failures (catastrophic failure) are not supposed to occur in RAID systems. Such systems are designed so disk failures are independent and the possibility that a second disk will fail before the data on a first failed disk can be reconstructed will be minimal. In order to shorten this susceptible period of "degraded" operation, a spare disk is frequently provided so the reconstruction of the failed disk can begin as soon as a failure is detected. Nevertheless, multiple disk failures do occur for a number of more or less unlikely reasons, such as a nearby lightning strike causing a power surge, or a physical tremor shaking the disks and disrupting the read/write heads over multiple disks. Such events can create logically invalid regions. This invention is equally useful for identifying logically invalid regions of disks whether the region in question is also physically bad.

Multiple disk failures may be classified in two categories: A. local or B. transient failures. Such failures stem from medium errors, localized hardware errors, such as corruption of track data, and bus errors. Type A and B errors are handled by retries. The retries are made automatically; the number of retries depends on the number of disks in the array and the demands on the system, including the errors detected in the other disks of the array.

C. Burst or severe errors. Such errors are seen over a large range of addresses or cause the disk to become inaccessible after an attempt is made to access a certain region. Type C errors are handled by failing a disk with powering down of the entire system. Type C errors are also referred to as "catastrophic" errors.

A system which is downed by a type C error is restored by the following steps. 1. The system is repowered. 2. An attempt is made to restore the failed disk through redundancy. 3. The failed disks are replaced and reconstructed.

Other classifications of failures have been proposed, for example, the following: 1. Transient failures. Unpredictable behavior of a disk for a short time. 2. Bad sector. A portion of a disk which cannot be read, often for physical reasons. 3. Controller failure. The disk contents are unaffected, but because of controller failure, the disk cannot be read. 4. Disk failure. The entire disk becomes unreadable, generally due to hardware faults such as a disk head crash. Pankaj Jalote, Fault Tolerance in Distributed systems, Prentice hall, Englewood Cliffs, N.J., 1994, pages 100–101.

Disk arrays which allow writeback-caching are subject to the danger of losing data which have been accepted from the host computer but which have not been written to the disk array. RAID-0 systems have no redundancy and no error protection. RAID 1–5 systems provide error correction for the loss of a single channel through parity methods. Error-detection in the event of multiple channel failure, however, cannot be guaranteed. Under these circumstances, data may be correctly written on some channels but not on others, a falsely valid parity might be returned, and corrupted data may be returned. If the unit must be powered down to correct the situation before the array can be brought back online, there may be no opportunity to rewrite the data successfully and live write-back data may also go unwritten.

Faulty cache memory may produce apparent multiple-disk errors of a persistent nature. For example, cache data with incorrect parity may generate bad SCSI parity on both the data channel and on the parity channel. In this case, when a write to disk is performed, two disks will report that the data are invalid.

The sharing of one bus between many disks, as is commonly done on RAID systems, creates a single point of failure in the bus which increases the probability of "two channel" failure. For example, in an array of five channels (four data channels and one parity channel) with each channel serving five disks, the failure of a single bus means than an error on any one of the 20 disks on the four other data channels will be unrecoverable. This has the same effect as a two channel failure.

In the present invention, a table of address ranges which have not been successfully written to a parity stripe is replicated on one disk on each channel in the array with frequent updating. After a catastrophic failure of multiple disks, assuming at least one of those disks can be written to, there will be a record of the failure on some disk. Since the record is on many disks, rather than only on the disk which experienced the failure, the controller can generate a list of all regions where data have been lost after the array has been repaired, even if the unit must be powered down before such a repair can be performed. This reduces the loss of down time for the system and reduces the cost of restoring the system.

The RAID Advisory Board has provided a summary of criteria for the classification of RAID systems with respect to reliability. Http://www.raid-advisory.com/EDAPDef.html. It is expected that the present invention will be useful in the development of "Failure Tolerant Disk Systems (FTDS) and Disaster Tolerant Disk Systems (DTDS).

U.S. Pat. No. 4,598,357 discloses a system in which data involved in a writeback error are reassigned to an unused portion of a working disk. The location of areas from which data have been lost are not recorded.

U.S. Pat. No. 4,945,535 discloses an address control device which, when it detects an error in a data word read

from a main memory device, changes the address of that error and does not use the memory area in subsequent data writes.

U.S. Pat. No. 5,166,936 discloses a method for automatically remapping a disk by removing a bad sector and replacing it with a good track of data. A flag is set during the process so that should power fail the process can be restarted.

U.S. Pat. No. 5,249,288 discloses an electronic printing system which identifies physically bad areas and remaps them through file allocation.

U.S. Pat. No. 5,271,012 discloses a RAID system tolerant to failure of two disks which uses the double generation of parity information using alternate rows and diagonals of direct access storage devices.

U.S. Pat. No. 5,274,799 discloses a RAID 5 system in which the copyback cache storage unit is used to store peak load data and completes the write function during relatively quiescent periods.

U.S. Pat. No. 5,285,451 discloses a mass memory system capable of tolerating two failed drives in which a number of disk drives are coupled to an equal number of buffers by X-bar switches. The switches couple and decouple functional and nonfunctional drives as necessary.

U.S. Pat. No. 5,412,661 discloses a data storage system in which disks are arrayed and each disk is controlled by two disk controllers. The system is tolerant of the failure of any one controller and has hot spare disks to accommodate disk failure.

U.S. Pat. No. 5,463,765 discloses a process in which invalid blocks of data are stored in a new location and used to recover the data of the faulty drive.

U.S. Pat. No. 5,479,611 discloses an error-correction technique in which data from a bad block on a disk are reassigned and reconstructed without the use of a cache memory.

U.S. Pat. No. 5,469,453 discloses a mass data storage apparatus in which bad blocks are time stamped and given a logical address. Comparison of the addresses and time stamps allows determination of failures of the writing devices.

U.S. Pat. No. 5,526,482 discloses a fault-tolerant storage device array in which at least two redundant copies of each pending data block are retained in the array controller's buffer memory and the copyback cache storage unit, providing protection against buffer failure.

U.S. Pat. No. 5,548,711 discloses a system including a DATA-RAM and a SHADOW-RAM. Write data from the CPU is stored in two independent memories to insure that pending Write data are not lost.

U.S. Pat. No. 5,564,011 discloses a non-RAID system in which critical data is replicated and used to regenerate failed control blocks.

U.S. Pat. No. 5,572,659 discloses an adapter for mirroring information on two channels which detects the failure of one channel and reads and writes from the other channel.

U.S. Pat. No. 5,574,856 discloses a storage device array in which data blocks of converted data are labeled with predetermined code bits which indicate the operation in which a fault occurs. In the presence of a fault, a data reconstruction operation and a data reassignment operation are indicated.

U.S. Pat. No. 5,574,882 discloses a system for identifying inconsistent parity in an array of storage in which a bit map of inconsistent parity groups is created.

U.S. Pat. No. 5,600,783 discloses a disc array system in which data for a faulty disc is stored in a cache until the disc is replaced.

U.S. Pat. No. 5,617,425 discloses an array supporting system in which drive controllers accept responsibility from the array controller for detecting write errors and reallocating data away from faulty discs.

U.S. Pat. No. 5,636,359 discloses a performance enhancement system which uses a directory means to prevent errors in the reading and writing of data.

U.S. Pat. No. 5,644,697 discloses a redundant array of disks in which the disks are divided into areas of varying size and having a single status table which indicates which areas are in use.

U.S. Pat. No. 5,657,439 discloses a system in which a logical region of a disk is used as a distributed spare for use in recovering data having errors.

Those prior art RAID systems tolerant to multiple disk failure exceeding the redundancy of the array depend on hardware, such as non-volatile memory or cache memory with a battery or extra disks, to cope with writeback cache loss in the event of multiple disk failure. The present invention uses only software and a small portion of reserved space on each disk to provide a reliable, inexpensive, widely applicable system for error-detection for write-back data lost during a catastrophic multiple disk failure.

BRIEF SUMMARY OF THE INVENTION

Catastrophic disk-array failures involve the failure of greater than one disk in a RAID 1–5 system, or any disk in a RAID 0 system. In almost all cases, however, there remains the ability of the controller to communicate with at least one disk in the array. The present invention uses software and a small portion of each disk in the array to write a bad area table on each disk. The bad area table provides the logical address and length of the area in the array's logical space which has been corrupted by physical damage on the media or other causes of write failure. After a catastrophic failure of multiple disks, assuming at least one disk can be written to, there will be a record of the failure on at least one disk. The record is on several disks, or at least one disk, rather than only on the disk which experienced the failure. The task of repairing the array is greatly simplified because all bad regions of the array can be easily identified. This reduces the loss of down time for the system and reduces the cost of restoring the system.

The process of writing failure records on one or more disks, as described in this patent, has the advantage of very rapidly recording the failure incident. It takes only about 20 milliseconds to record to a disk. Recording to flash memory can require a significantly longer time, up to several seconds in the worst case. The difference in recording time may be crucial under certain failure conditions, for example, in the case of a power failure recording to disk could be accomplished while recording to flash memory would fail.

This invention is a process for designating physically or logically invalid regions of storage units as a whole or fractional number of blocks on storage units on which data has been striped, in a fault-tolerant storage device array of a number of failure independent storage units which receive information from a writeback cache and a controller with a writeback cache. First, the logical address and length of the physically or logically invalid region is determined. Second, the address and length is written on a bad region table, and thirdly, the bad table region is replicated on each storage unit. The process may be used with storage units which are

disks, tapes, or CDS which are connected to the controller. The process may be used when the bad region is due to a writeback, a read, or a write error, and in a system in which a number of storage units exceeding the redundancy have failed, or when the data is being restored or replaced on a spare storage unit or in a non-redundantly configured array. In addition to being replicated on two or more storage units, the bad region table may also be replicated in volatile memory with battery backup in the controller, or on additional storage units separate from the array of storage units. Finally, the process can include the steps of time-stamping entries and determining the most recent entry.

This invention is especially useful in connection with a host computer with a RAID system which is periodically backed up to tape and which participates in a distributed system through a network. In the absence of this invention, corrupted data could be sent to the host and then propagated through the network to other nodes in the distributed system. With our invention, the data would be recognized as invalid or lost by the host system, and there would exist no danger of corrupted data leaving the local node or being used for processing by the local node. The backup tape would then be used to roll the node back to an earlier, consistent configuration.

This invention is also especially useful in a system with two RAID arrays software-mirrored by the host computer's operating system. After the occurrence of a catastrophic system crash, the host could reassemble all data which is correct on either of the two arrays, using the present invention. Without this invention, the host could not identify which blocks were in error on which RAID device.

The objective of this invention is to provide an inexpensive means for identifying the locus of catastrophic failure of RAID 0–5 systems.

Another objective is to provide means for rapid identification of failed areas in a RAID 0 system with no redundancy.

Another objective is to provide means for rapid identification of failed areas in RAID 2–5 systems which utilize parity to correct single disk failures.

Another objective is to provide means for inexpensive catastrophic failure identification which do not require hardware other than small areas of the storage disks.

Another objective is to provide software means widely adaptable to a variety of configurations of RAID 0–5 systems for identification of the sites of catastrophic failures.

A final objective is to facilitate the rapid recovery of RAID 0–5 systems from catastrophic failure occasioned by physical or logical sources.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS.

FIG. **1** is a schematic of the external view of the array, disk array controller, and host computer.

FIG. **2** is a schematic of the system architecture showing only one channel.

FIG. **3** is a schematic of the DMA sync hardware.

FIG. **4** is a flowchart of the portion of the system boot process relevant to the present invention.

FIGS. **5***a* and **5***b* are flowcharts of the routine which is invoked periodically to update the bad block tables on the disks.

FIG. **6** is a flowchart of the routine which is invoked prior to a data operation on the disk array.

FIG. **7** is a flowchart of the subroutine which checks the validity of data loaded during a blocking/deblocking operation to perform read-modify-write to the disks.

FIG. **8** is a flowchart of the routine which is invoked when an operation to the disks completes with bad status.

FIG. **9** is a flowchart of the subroutine which checks a given range of logical blocks for overlap with the invalid regions logged in the bad block table.

FIG. **10** is a flowchart of the subroutine which adds a range of logical blocks to the bad block table.

FIG. **11** is a flowchart of the subroutine which deletes a range of logical blocks from the bad block table.

FIG. **12** is a flowchart of the process performed when a storage unit reports a medium error status on a read.

FIG. **13** is a flowchart of the change range subroutine.

FIGS. **14***a* and **14***b* are flowcharts of the subroutine called when a region of the array is determined to be invalid.

FIGS. **15***a* and **15***b* are flowcharts of the subroutine called when a write operation successfully commits data to the storage array.

DETAILED DESCRIPTION OF THE INVENTION.

FIG. **1** is a schematic of the external view of a RAID 3 system comprising a single host computer, a RAID controller, and two tiers of 5 Direct Access Storage Device (DASD) units with two parity DASDs and two additional hot-spare DASDs which incorporates the invention. All the DASDs in a system taken as a whole is referred to as an "array" of DASDs. A group of DASDs served by separate channels across which data is striped is referred to as a "tier" of DASDs. A DASD may be uniquely identified by a channel number and a tier letter, for example DASD 1A is the first disk connected to channel 1 of the controller.

A preferred controller is the Z-9100 Ultra-Wide SCSI RAID controller manufactured by Digi-Data Corporation, Jessup Md.

The host computer **10** is connected by the host small computer system interface (SCSI) bus **20** to the disk array controller **30**. Disk array controller **30** is connected to DASD IA **40** and DASD 1B **41** via the channel **1** disk SCSI data bus **70**; to DASD 2A **42** and DASD 2B **43** via the channel **2** disk SCSI data bus **71**; to DASD 3A **44** and DASD 3B **45** via the channel **3** disk SCSI data bus **72**; and to DASD 4A **46** and DASD 4B **47** via the channel **4** disk SCSI data bus **73**; respectively. Parity DASD 5A **50** and 5B **51** are connected to the Disk Array Controller **30** by the channel **5** SCSI parity disk bus **80**. Spare DASD 6A **60** and 6B **61** are connected to Disk Array Controller **30** by the channel **6** SCSI hot spare disk bus **90**.

Additional tiers of DASDs may be used. Additional host channels and host computers may be used on the system.

Any suitable host computer may be used.

FIG. **2** is a schematic of the system architecture of the disk array controller (**30** in FIG. **1**) showing one disk channel and one host channel only. The flow of data between host and disk array is indicated by the heavy line. Data is received from the host computer via the host SCSI bus **20** into the SCSI input/output processor (SCSI IOP) **31**. The SCSI IOP initiates memory transactions to or from the cache memory **32** through the bridge chip **33** which bridges the system bus and the cache bus. A cache bus **22** connects the bridge chip **33**, cache memory **32**, and the hardware control mechanism DMA Sync **34**. The DMA Sync acts as a direct memory

access (DMA) controller with the additional RAID-3 function of parity generation and checking and replacement of data with a hot spare. It also generates reads or writes to specific cache addresses and translates the data between the cache bus **22** and the SCSI interface chip **35** on the individual channel. The DMA Sync also controls the necessary hardware handshaking signals for direct memory access (DMA). Although only one SCSI interface chip **35** and SCSI disk bus **70** is shown in FIG. **2**, there are as many of these components as there are busses of DASDs. The SCSI interface chip **35** is connected by connector **24** to bridge chip **33**. The DMA Sync **34** is connected by connector **23** to the bridge chip **33**. The non-volatile flash electrically erasable programmable read-only memory (EEPROM) **39** stores parameters for the controller and the system firmware, which is uncompressed from the flash into 4 Megabyte local dynamic random-access memory (DRAM) **38** when the system is booted. A DUART chip **37** has two RS-232 connectors **27** which allow the user to configure the unit through a command line interface and also provide communication between the unit and a workstation running debugging software. The flash memory **39**, local DRAM **38**, DUART chip **37** and CPU **36** and system bus **21** are connected by a processor bus **25**. Both the SCSI interface chip **35** and the DMA Sync **34** are programmed by the CPU **36** through the system bus **21** and the bridge chip **33**.

A preferred CPU **36** is an Intel 960RP available from Intel Corporation, Santa Clara, Calif. A preferred SCSI IOP **31** is a Symbios SYM53C875 available from Symbios Logic Incorporated, Colorado springs, Colo. A preferred system bus **21** is a 32-bit bus designed in accordance with the Peripheral Controller Interconnect (PCI) specification. A preferred SCSI interface chip **35** is a QLOGIC FAS366U Ultra Wide SCSI interface chip available from QLogic Corporation, Costa Mesa, Calif.

FIG. **3** is a schematic of the DMA sync hardware. The controlling state-machine and related signals and registers, including handshaking signals, are omitted from this schematic. Data enter and exit the DMA sync (**34** in FIG. **2**) via the cache bus **22** and the individual channel interface chips in the SCSI Interface **35**. Data enter and exit the DMA sync (**34** in FIG. **2**) via the cache bus **22** and data buses **110**, **120**, **130**, **140**, **150**, **160** to the individual channel interface chips **35**, **92**, **93**, **94**, **95**, **96**. During a write from cache to disk, data from the cache bus is latched in two 36 bit registers **310**, **312** from which a parity byte is generated by parity-generating circuitry **320**. The eight data bytes and two parity bytes are latched in two sets of five 9 bit registers **330**, **332**. The data are then rearranged to be byte-striped across the disks and are passed to a multiplexor **340** to provide data for the spare channel if it is to be used. For reads from disk to cache the process is reversed.

FIG. **4** is a diagram of the data structure used for bad block management, herein referred to as the BBM MAP, which is maintained in the controller's memory. The BBM MAP consists of an array of individual records called the BBM TABLE **402**. Each record describes a region which has been determined to be invalid. The fields contained in the record have the following meanings. LUN refers to the SCSI logical unit through which the host computer would access the invalid region. START ADDRESS indicates at what logical block address within the LUN the invalid region starts. EXTENT indicates for how many logical blocks from START ADDRESS the invalid region extends. The pointer NEXT is used as a linkage field so that the elements within the BBM TABLE **402** can be organized into linked lists. Additionally, the BBM MAP contains BBM FIRST FREE **404**, a pointer to the first unused entry in the BBM TABLE, BBM LIST **406**, a pointer to the first entry in the BBM TABLE **402** which describes an invalid region, and BBM DIRTY **408**, a boolean value which indicates whether the table has been altered since it was last saved to the disk array. The unused entries in the BBM TABLE **402** are organized into a linked list of which BBM FIRST FREE **404** is the head, and the entries describing invalid regions are organized into a linked list of which BBM LIST **406** is the head. In FIG. **4**, the table is shown as containing two unused entries and two used entries.

In the discussions which follow, the following notation for the comparison of entries in the BBM TABLE **402** will be employed to simplify the discussion. When used to compare entries for which the value of the LUN field is identical, START(X) will be taken to mean the value of an entry's START ADDRESS field, where X refers to the entry in question. Similarly, END(X) will be taken to mean the sum of the entry's START ADDRESS and EXTENT fields minus one, which is the last logical block address described by the entry as invalid. In comparisons between entries describing regions of different LUNs, it is defined that END(Y)+1<START(Z) for any regions Y and Z for which the value of entry Y's LUN field is less than the value of entry Z's LUN field. In this way, the address spaces of all of the LUNs represented by the controller unit are flattened into a single, larger address space for purposes of ordering and comparison. The notation NEXT(X) will be taken to mean the entry referred to by entry X's NEXT field.

Since the maintenance of linked lists through insert and delete operations and the use of free lists to dynamically allocate elements from within a static structure are well known in the art, these will not be further elaborated. In the discussion which follows, operations which attempt to allocate an unused entry from the BBM TABLE **402**, returning a failure code if none is available, and which return an entry no longer needed to the pool of free entries will be assumed, as will an operation to sequence the linked list structure prior to saving to disk and an operation to restore the original structure from an image on disk. The product of this operation, to which are appended a timestamp and a marker to indicate that the data is thus formatted, will be referred to as a BBM STAMP. All of these processes are well known in the art. Within the linked list BBM LIST **406** the additional property is preserved that for each entry A which is followed by entry B in the list, END(A)+1<START(B), which implies that the entries are ordered, that they do not overlap and that they are noncontiguous.

FIGS. **5***a* and **5***b* depict the read-modify-write process known as blocking/deblocking, performed prior to certain write operations which a host computer may initiate by a controller which byte-stripes data onto a set of disks. Many devices, including host adapter cards and disks, support only a fixed size for logical blocks, which is the minimum unit of data which can be transferred to or from the device. For SCSI devices, this fixed size is frequently 512 bytes. Unless the host computer supports a block-size which is a multiple of the block-size supported by the storage units in the array multiplied by the number of storage units across which the data is to be striped, the possibility exists for the host computer to request a write which alters only a fractional portion of a disk block. In this case, the controller must read the data which is recorded on the block in question prior to performing the write in order to avoid corrupting the data which shares the same blocks in the storage array with the host data to be written.

In FIG. **5***a*, a single block of data **502** is written by the host computer to the controller and placed in the controller's

cache memory **504**. Also shown in memory are three adjacent blocks **508, 510, 512** which together with **502** map onto the same set of four blocks **514, 516, 518, 520** on four different devices within the storage array **506**. Such blocks with logical addresses preceding the host data to be written **508, 510** are referred to as a blocking/deblocking head (BDB head), and such blocks with logical addresses subsequent to the host data **512** are referred to as a blocking/deblocking tail (BDB tail). Arrow **522** depicts the process of host data being written into cache memory **504**. Arrow **524** depicts the blocking/deblocking read of byte-striped data from the storage array into the cache memory **504**. The portion of the data which corresponds to the block of host data **502** is shown in a dotted line to indicate that the data being transferred from the storage array **506** is blocked from overwriting the host data. In this way, the data from the host computer is merged with the other data which share the same set of blocks within the storage array **506**. Arrow **526** depicts the combined data being written back to the storage array **506**.

FIG. **5***b* shows the effect of an unrecoverable physical medium error on one block in the storage array during the blocking/deblocking operation of FIG. **5***a*, assuming no operational parity disk is available. In this case, the only logically valid area of the blocking/deblocking region after the operation is the data written by the host, since the portions of blocks **508, 510** and **512** which were stored in block **520** have been overwritten with invalid data and the data from those blocks which resides on disk blocks **512, 516** and **518** are not sufficient to reconstruct the entirety of the lost blocks.

FIG. **6** is a flowchart of the steps pertaining to bad block management taken when the system boots. Block **602** is the entry point for the routine. In block **604** the system allocates memory for the BBM MAP structure and initializes it to contain no bad regions. Variables used to find a valid stamp in the disk array are initialized in block **604**. The loop control variable DISK is set to indicate the first disk in the array, and a variable CANONICAL STAMP is set to a null value. In block **608** a stamp containing the bad block table is read from the disk referred to by DISK into a local buffer BBM STAMP, which is checked for a valid stamp format in block **610**. If BBM STAMP is determined to be invalid, control passes to block **616** where the presence of other disks is checked. If there is another disk, it is assigned to DISK in block **618** and the loop repeats. If there is no other disk, control passes out of the loop to block **620**. If BBM STAMP is valid in block **610**, its timestamp is checked against the timestamp of CANONICAL STAMP (where the timestamp for a null stamp is defined to be older than any valid timestamp) in block **612**. If it is more recent than CANONICAL STAMP, CANONICAL STAMP is set to BBM STAMP. Control then passes to block **616** for the next iteration of the loop. When there are no more disks to check, control passes to block **620**, where CANONICAL STAMP is checked for a null value. If it is not null, then an appropriate stamp has been located, and all the bad regions described in CANONICAL STAMP are added to the BBM MAP in block **622**. This portion of the boot-up process is completed in block **624**.

FIG. **7** is a flowchart of the process CHECK BBM FLUSH which is performed periodically by the system tasks running on the CPU such that the process is guaranteed to be performed by a system task which alters the BBM MAP structure soon after that alteration is made. The process starts in block **702**. In block **704**, the BBM DIRTY bit of the BBM MAP structure is checked. If it is not set, the process completes in block **712**. If the bit is set, a valid BBM STAMP is built in the controller's local memory containing the timestamp for the current time and all of the entries from the BBM TABLE which are linked in the BBM LIST in block **706**. In block **708**, write operations of the BBM STAMP onto a specific, reserved address of each disk in the array are initiated. The BBM DIRTY bit is then cleared in block **710** to indicate that the alterations to the map have been saved. The process then completes in block **712**.

FIG. **8** is a flowchart of the process CHECK TRANSFER which is performed prior to any operation which accesses or alters data stored in the array. The process starts in block **802**. In block **804**, the direction of the operation is checked. If the operation writes data to the storage array, control is transferred to block **812**, and the process completes.

For read operations, control is transferred to block **806**, and the source of the command for the data transfer is checked. If the command originated from the host computer, control passes to block **808**. Otherwise, the command is internally generated by the controller (such as the reconstruction of a disk or parity verification of a healthy array), and control passes to block **814**, where the subroutine CHECK RANGE is called for the range of logical addresses described by the command. If no invalid blocks are present in the range, control is transferred to block **812**, and the process completes. If there are invalid blocks in the range, control passes to block **834**, where the region of the controller's cache memory which would receive the remainder of the data starting from the first invalid block is marked as invalid in order to prevent the invalid data from being transferred to the host as a cache hit. A means for selective invalidation of cache blocks is assumed, since the management of cache memory, including the invalidation of specific regions of a cache, is outside of the scope of the present invention. After cache invalidation is performed, control passes to block **812**, and the process completes.

For read operations originating from the host computer, control passes to block **808**, where it is checked whether the operation in question is a blocking/deblocking read operation to be performed before a host write. If so, control is transferred to the subroutine CHECK BDB in block **810**, where the blocks loaded during the blocking/deblocking process are individually checked for validity and invalidated in cache if necessary. Control then passes to block **812**, and the process completes.

For non-blocking/deblocking read operations originating from the host computer, control passes to block **819**, where the subroutine CHECK BDB is called for the blocking/deblocking head, if any, preceding the requested host data to prevent any invalid data from being transferred to the host as a cache hit. In block **820**, the subroutine CHECK RANGE is called for the range of logical addresses to be read from the storage array. This range may include a blocking/deblocking tail which will be loaded into memory along with the requested data. It may also include a lookahead, a number of blocks to be loaded sequentially after those blocks which the host requested. The technique of lookahead is well known in the art to enhance performance for hosts which are likely to access data from a storage system sequentially. In block **822** it is checked whether any of the data described by the command contains invalid blocks. If not, control passes to block **812**, and the process completes. If there are invalid blocks in the range, control passes through connector **824** to block **828**, where it is checked whether any of the data requested by the host is invalid. If not, control is transferred to block **826**, and all of the requested data is transferred to the host, after which the invalid portion of the command is invalidated in the cache

memory to prevent the invalid data from being transferred to the host as a cache hit.

If the invalid blocks are determined to fall within the portion of the command requested by the host, control is transferred to block **836**, where it is checked whether the first block of data requested by the host is registered as invalid. If so, control passes to block **838**, where the entire command is invalidated in the cache memory to prevent the invalid data from being transferred to the host as a cache hit. A MEDIUM ERROR status is then sent to the host in block **840**, indicating the requested address as the location of the block in error. If some of the data requested by the host is valid, control passes to block **830** where all of the valid data requested is transferred to the host. In block **832**, A MEDIUM ERROR status is then sent to the host indicating the first invalid address as the location of the block in error. Control then passes to block **812**, and the process completes.

FIG. **9** is a flowchart of the subroutine CHECK BDB, which checks for invalid blocks loaded during a blocking/deblocking read operation. The process begins in block **902**. In block **904**, the variable BLOCK is set to the first logical block address which will be loaded by the operation. In block **906**, the subroutine CHECK RANGE is called for the range of a single block at the address BLOCK. If the block is labeled as invalid in block **908**, it is invalidated in cache in block **914**. In block **910**, BLOCK is set to the next block which will be loaded by the blocking/deblocking operation. If there are more blocks to load in block **912**, control passes to block **906** for the next iteration of the loop. Otherwise, control passes to block **916**, and the subroutine completes.

FIG. **10** is a flowchart of the process which is performed when a write operation to the storage array has been completed, successfully or unsuccessfully, by the controller. In the case of a failure, this process will only be called after the number of retries prescribed by the errorhandling policies of the controller have been performed. If the array has redundancy and errors occur on only one disk, the disk may be removed from operation prior to the invocation of this process, in which case the status of the write operation will be considered to be good. The process begins in block **1002**. In block **1004**, the status of the operation is checked. If the data have been successfully committed to all of the storage units involved, control passes to block **1012**, where the subroutine DELETE BAD BLOCKS is invoked for the range of logical addresses described by the host command (not including blocking/deblocking heads or tails), indicating that the range of addresses contain valid host data. The subroutine then completes in block **1010**. If the data was not successfully committed to the array even after retries and/or the removal of a redundant disk from operation, control passes to block **1006**, where the subroutine ADD BAD BLOCKS is called for the range of logical addresses described by the command (including blocking/deblocking heads or tails). If the operation is determined to be a writethrough operation in block **1008**, sense indicating the failure of the write is sent to the host in block **1014**. If the operation is a write-back operation, the host cannot be immediately informed of the error, so control passes to block **1010** and the process completes.

FIG. **11** is a flowchart of the process which is performed when a blocking/deblocking read operation completes with a failure status after the prescribed number of retries. By invalidating the blocking/deblocking data rather than aborting the associated write command issued by the host computer, this routine favors data from the host over the data sharing the same physical blocks in the storage array. This policy is based on the fact that the write data from the host is known to contain data which the host uses, whereas the blocking/deblocking data may be in unused space. The process begins in block **1102**. In block **1104**, BLOCK is set to the logical address of the first logical block which would have been loaded by the blocking/deblocking operation. A call is made to the subroutine ADD BAD BLOCKS in block **1106** to add that block to the BBM MAP. In block **1108**, BLOCK is set to the logical address of the next block which would have been loaded, and if there is such a block, control passes through block **1110** to block **1106** for another iteration of the loop. When there are no more blocks, control passes to block **1112**, and the process completes.

FIG. **12** is a flowchart of the process which is performed when a storage unit reports a MEDIUM ERROR status on a read. If the array is not in a redundant state, this process will be called in the case of failure only after the prescribed number of retries have expired. Since the DMA sync hardware automatically rebuilds the lost data into cache from parity without any additional read or parity-building operations, no retries need be performed when the array is in a redundant state. Additionally, the block containing physically flawed media may be remapped to a different location, either automatically by the storage device or using the SCSI REASSIGN BLOCKS (0x07) command. The process begins in block **1202**. In block **1204**, it is determined what range of logical addresses map into the disk block for which the medium error occurred. If the data is determined to have been successfully reconstructed by the DMA sync hardware in block **1206**, writeback bits are set in the cache node descriptor corresponding to the logical address range where the error occurred and a writeback flush operation is queued for the cache node in order to write back the corrected data to the storage unit which reported the failure. If the data was not reconstructed, control passes to block **1210** where the subroutine ADD BAD BLOCKS is called to register the affected blocks as invalid in the BBM MAP. If the operation is determined to be a reconstruct in block **1212**, it is given a RETRY status in block **1214**. Since reconstruction often crosses large extents which have never been used by the host computer and which are likely to contain medium errors since they have not recently been written, and since the array can only be restored to redundancy by the completion of a reconstruction, it is desirable to allow many such reassignments to occur before aborting the operation. If the operation is determined to be a blocking/deblocking read in block **1212**, control passes to block **1220** and the error, which has been documented in the BBM MAP, is ignored, allowing the associated host write to proceed. Other operations are given an ABORT status in block **1216**, and the steps prescribed by the controller's error-handling policy for aborted commands are taken, including the sending of sense describing the error to the host, when appropriate.

FIG. **13** is a flowchart of the CHECK RANGE subroutine, which determines whether any logical blocks within a given range on a given LUN are listed as invalid in the BBM MAP and, if so, returns the address of the first invalid block in the range. The subroutine begins in block **1302**. In block **1304**, the BBM LIST is assigned to the local pointer variable REGION. If REGION is null, the range to be checked does not overlap any bad blocks and the subroutine returns null in block **1308**. If REGION is non-null, control passes to block **1310** where the first block of the invalid region designated by REGION is compared to the last block of the region to be checked, designated by XFR_{13}REGION. There can be no overlap if START(REGION)>END(XFR_{13}REGION), in which case control passes to block **1312**, where the pointer

REGION is set to the next entry in the BBM LIST, and then to block **1306** for another iteration of the loop. If START (REGION)<=END(XFR_{13}REGION), there is a possibility of overlap, so the last block of REGION is compared with the first block of XFR_{13}REGION in block **1314**. If END (REGION) >=START(XFR_{13}REGION), the two regions overlap, so the greater of START(REGION) and START (XFR_{13}REGION) is returned by the subroutine in block **1316** as the first invalid block in the given range. Otherwise, there is no overlap, and control passes to block **1312** to examine the next entry in the list.

FIG. **14** is a flowchart of the subroutine ADD BAD BLOCKS, which is called when a region of the array is determined to be invalid. The subroutine begins in block **1402** of FIG. 14a. In block **1404**, the bit BBM DIRTY is set to indicate that a write of the BBM MAP to disk will need to occur. In block **1406**, the BBM LIST is assigned to the local pointer variable REGION. If REGION is null in block **1408**, the range does not precede or overlap any entries in the BBM LIST and is inserted at the end of the list in block **1410**. The subroutine then completes in block **1412**. If REGION is non-null, the first block of REGION is compared to the block after the last block of the region to be added, designated as NEW_{13}REGION, in block **1414**. If START(REGION)>1+END(NEW_{13}REGION), then NEW_{13}REGION precedes REGION, and the two do not need to be merged into a single descriptor. Control passes to block **1416**, where the descriptor for NEW_{13}REGION is inserted in front of REGION in the BBM LIST. The subroutine then completes in block **1412**. Otherwise, the block after the last block of REGION is compared with the first block of NEW_{13}REGION. If END(REGION)+1<START (NEW_{13}REGION), then REGION precedes NEW_{13}REGION and the two do not need to be merged into a single descriptor. Control then passes to block **1420**, where REGION is set to the next entry in the BBM LIST, and then to block **1408** for the next iteration of the loop.

In the case where descriptors need to be merged, control passes through connector **1422** to block **1424** of FIG. **14***b*, where a new local pointer variable, $MERGE_{13}$REGION, which is used to determine whether any entries in the BBM TABLE are entirely contained within NEW_{13}REGION and must be deleted, is initialized to point to the next entry after REGION in the BBM LIST. Because the list is ordered, $MERGE_{13}$REGION is known to start after the first block of NEW_{13}REGION. In block **1426**, the first block of $MERGE_{13}$REGION is compared to the block after the last block of NEW_{13}REGION. If START($MERGE_{13}$REGION) <=1+END(NEW_{13}REGION), $MERGE_{13}$REGION is entirely contained within NEW_{13}REGION and is merged with REGION in block **1432**. The descriptor of $MERGE_{13}$REGION can then be returned to the list of free descriptors in block **1434**, and the pointer is advanced to the next entry in the list in block **1436**. This process repeats until $MERGE_{13}$REGION is determined to be disjoint from NEW_{13}REGION in block **1426**.

When all descriptors contained within NEW_{13}REGION have been merged, control passes to block **1428**, where the first block of REGION is set to be the lesser of the first block of REGION and the first block of NEW_{13}REGION. In block **1430**, the last block of REGION is set to be the greater of the last block of REGION, which in the loop in the preceding paragraph will have been set to the last block of the last region to be merged, and the last block of NEW_{13}REGION. The subroutine then completes in block **1412**.

FIG. **15** is a flowchart of the subroutine DELETE BAD BLOCKS, which is called when a write operation successfully commits data to the storage array. The subroutine begins in block **1502** of FIG. **15***a*. In block **1504**, the BBM LIST is assigned to the local pointer variable REGION. If REGION is null in block **1506**, the range of blocks is not listed in the BBM MAP, so the subroutine completes in block **1508**. If REGION is non-null, but in block **1510** the first block of REGION succeeds the last block of the region to be deleted, designated as XFR_{13}REGION, there can be no more entries in the list to be deleted, and the subroutine completes in block **1508**. Otherwise, the last block of REGION is then compared to the first block of the region to be deleted in block **1512**. If END(REGION)<START (XFR_{13}REGION), the regions are disjoint, and control passes to block **1518**, where REGION is set to the next entry in the BBM LIST, and then to block **1506** for the next iteration of the loop. Otherwise, there is overlap, and part or all of REGION must be deleted. In this case, the BBM DIRTY bit is set in block **1514** to indicate that the BBM MAP will need to be saved to disk and control passes through connector **1516** to block **1522**.

The flowchart of FIG. **15***b* shows the various cases of deletion of the portions of REGION which overlap XFR_{13}REGION. In block **1522**, it is checked whether REGION both ends before the end of XFR_{13}REGION and begins after the beginning of XFR_{13}REGION. In this case, REGION is entirely contained within XFR_{13}REGION, and its descriptor is deallocated in block **1524**. Control then passes through connector **1520** back to the main loop to check the next entry in the BBM LIST. In block **1526**, it is checked whether REGION ends before the end of XFR_{13}REGION but starts before XFR_{13}REGION. In this case, there are blocks at the head of REGION which are still invalid, so the end of REGION is set to the block before the first block of XFR_{13}REGION. Control then passes through connector **1520** back to the main loop to check the next entry in the BBM LIST. In block **1530**, it is checked whether REGION begins after the beginning of XFR_{13}REGION but ends after the last block of XFR_{13}REGION. In this case, there are blocks at the tail of REGION which are still invalid, so the start of REGION is set to the block after the last block of XFR_{13}REGION.

If none of the above are the case, XFR_{13}REGION must be contained within REGION. In this case the descriptor of REGION should be broken in two. A new descriptor is allocated in block **1534**. If no descriptor is available in block **1536**, REGION is truncated to end at the block before XFR_{13}REGION in block **1542**. If the allocation is successful, NEW_{13}REGION is set to begin from the block after the end of XFR_{13}REGION and end at the end of REGION in block **1538**. NEW_{13}REGION is then inserted in the list after REGION in block **1540**. REGION is then truncated to end at the block before XFR_{13}REGION in block **1542**. The subroutine then completes in block **1508**.

It will be apparent to those skilled in the art that the examples and embodiments described herein are by way of illustration and not of limitation, and that other examples may be used without departing from the spirit and scope of the present invention, as set forth in the claims.

I claim:

1. A method for designating physically or logically invalid regions of storage units in a fault-tolerant storage device array comprising a plurality of failure independent storage units for storing information which receive information from a writeback-cache, and a controller having a writeback-cache comprising the steps:

a. determining the logical address and length of each physically or logically invalid region,

b. writing the logical address and length of each physically or logically invalid region on a bad region table, and

c. replicating the bad region table on two or more but less than all of the storage units.

2. A method for designating physically or logically invalid regions of storage units in a fault-tolerant storage device array comprising a plurality of failure independent storage units for storing information which receive information from a writeback-cache, and a controller having a writeback-cache comprising the steps:

a. determining the logical address and length of each physically or logically invalid region,

b. writing the logical address and length of each physically or logically invalid region on a bad region table, and

c. replicating the bad region table on stable storage units separate from the array of storage units.

3. A method for designating as invalid either a whole or fractional number of blocks or regions on a plurality of storage units across which data has been striped after physical error or corruption on a storage unit or storage units in the plurality have occurred in a fault-tolerant storage device array comprising a plurality of failure independent storage units for storing information which receive information from a writeback cache, and a controller having a writeback-cache comprising the steps:

a. determining the physical address and length of each block or region of physical error or corruption,

b. determining the set of logical blocks which map onto the region of physical corruption,

c. determining the subset of the logical blocks from step b which are made logically invalid due to the physical error or corruption, and

d. replicating the bad region table on two or more but less than all of the storage units.

4. A method for designating as invalid either a whole or fractional number of blocks or regions on a plurality of storage units across which data has been striped after physical error or corruption on a storage unit or storage units in the plurality have occurred in a faulttolerant storage device array comprising a plurality of failure independent storage units for storing information which receive information from a writeback cache, and a controller having a writeback-cache comprising the steps:

a. determining the physical address and length of each block or region of physical error or corruption,

b. determining the set of logical blocks which map onto the region of physical corruption,

c. determining the subset of the logical blocks from step b which are made logically invalid due to the physical error or corruption,

d. recording the logical address and length of all logically invalid ranges of blocks, and

e. replicating the bad region table on stable storage units separate from the array of storage units.

* * * * *

US 20020016942A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2002/0016942 A1

MacLaren et al. (43) Pub. Date: Feb. 7, 2002

(54) HARD/SOFT ERROR DETECTION

(76) Inventors: John M. MacLaren, Cypress, TX (US); Tim Majni, The Woodlands, TX (US)

Correspondence Address:
Michael G. Fletcher
Fletcher, Yoder & Van Someren
P.O. Box 692289
Houston, TX 77269-2289 (US)

(21) Appl. No.: 09/966,891

(22) Filed: Sep. 28, 2001

Related U.S. Application Data

(63) Continuation of application No. 09/769,958, filed on Jan. 25, 2001, which is a non-provisional of provisional application No. 60/178,108, filed on Jan. 26, 2000.

Publication Classification

(51) Int. Cl.[7] G11C 29/00

(52) U.S. Cl. 714/718

(57) ABSTRACT

A system and technique for detecting and classifying data errors in a memory device. More specifically, hard and soft data errors in a memory device are detected by initiating a READ request initiated from a host controller. If an error is detected, the data is corrected and re-written to the corresponding memory location. A second READ request is then issued to read the corrected data. If a second error is detected in the corrected data, the error is classified as a hard error and a counter is incremented to track the number of hard errors detected in the system Once a programmable threshold number of hard errors are detected in a particular memory segment, an indicator, such as a light emitting diode (LED), is used to indicate that the corresponding memory segment should be replaced.

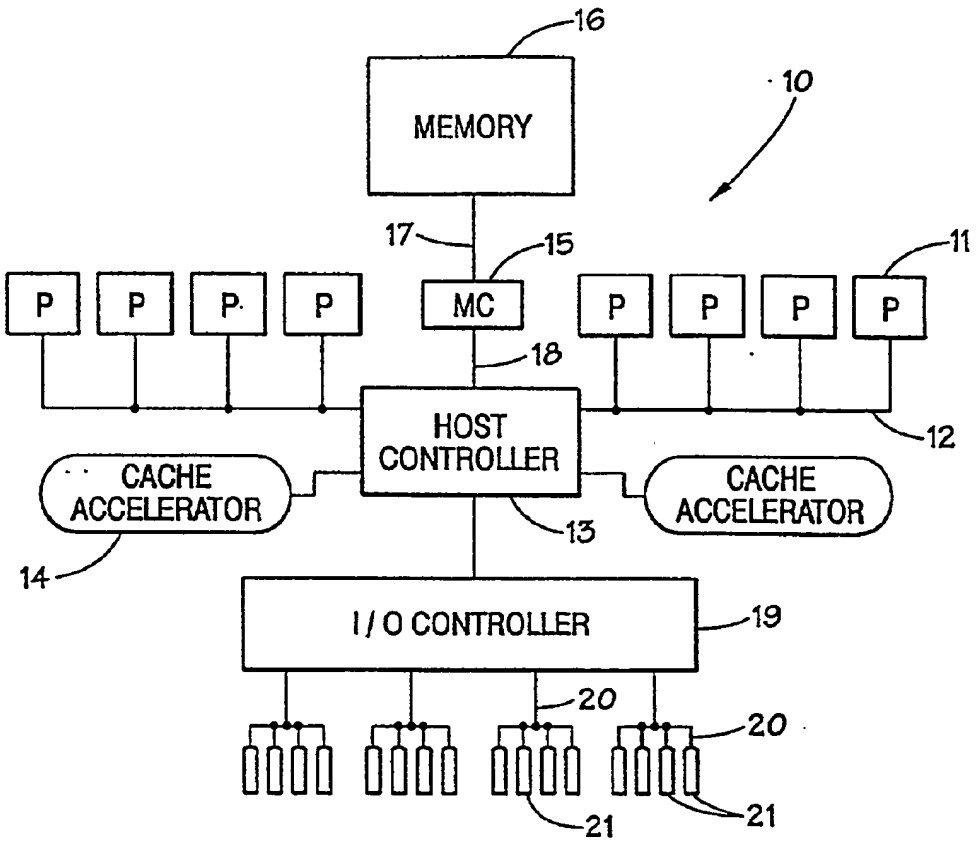
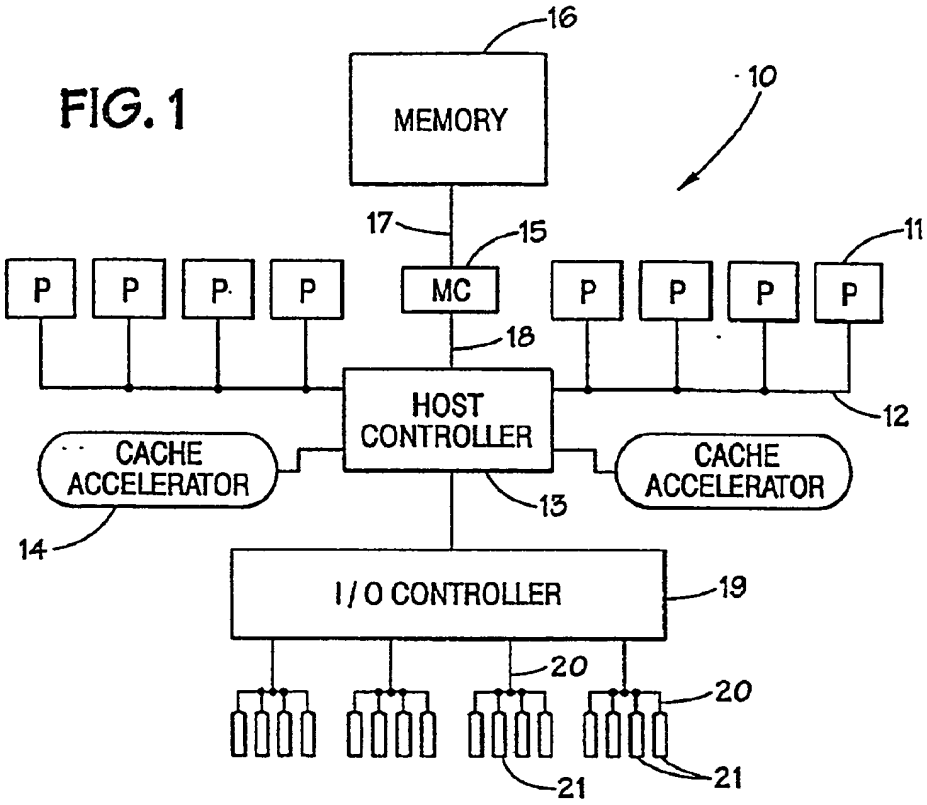


FIG. 1


16
MEMORY
10
17
15
11
P
P
P
P
MC
P
P
P
P
18
HOST CONTROLLER
12
CACHE ACCELERATOR
CACHE ACCELERATOR
13
14
I/O CONTROLLER
19
20
20
21
21

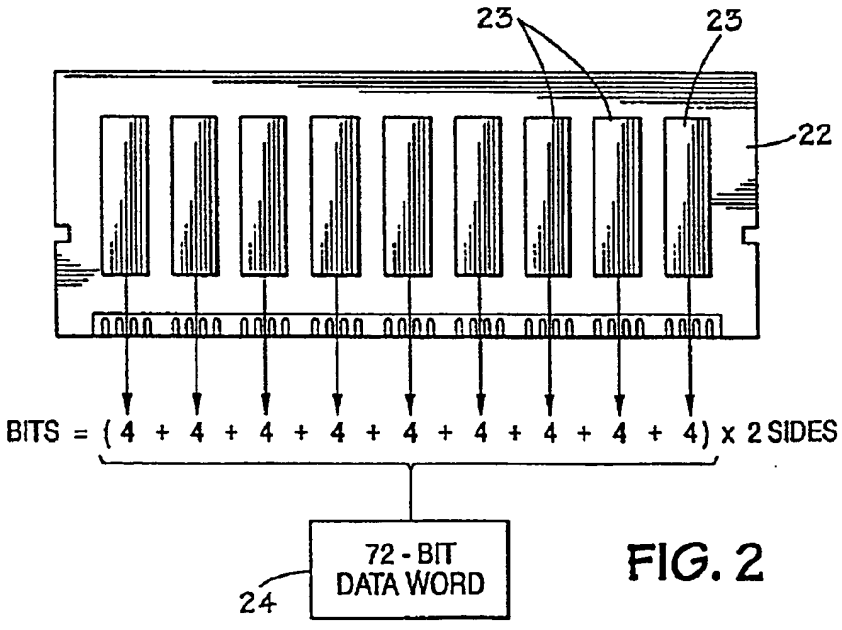

23
23
22
BITS = (4 + 4 + 4 + 4 + 4 + 4 + 4 + 4 + 4) x 2 SIDES
72 - BIT DATA WORD
24

FIG. 2

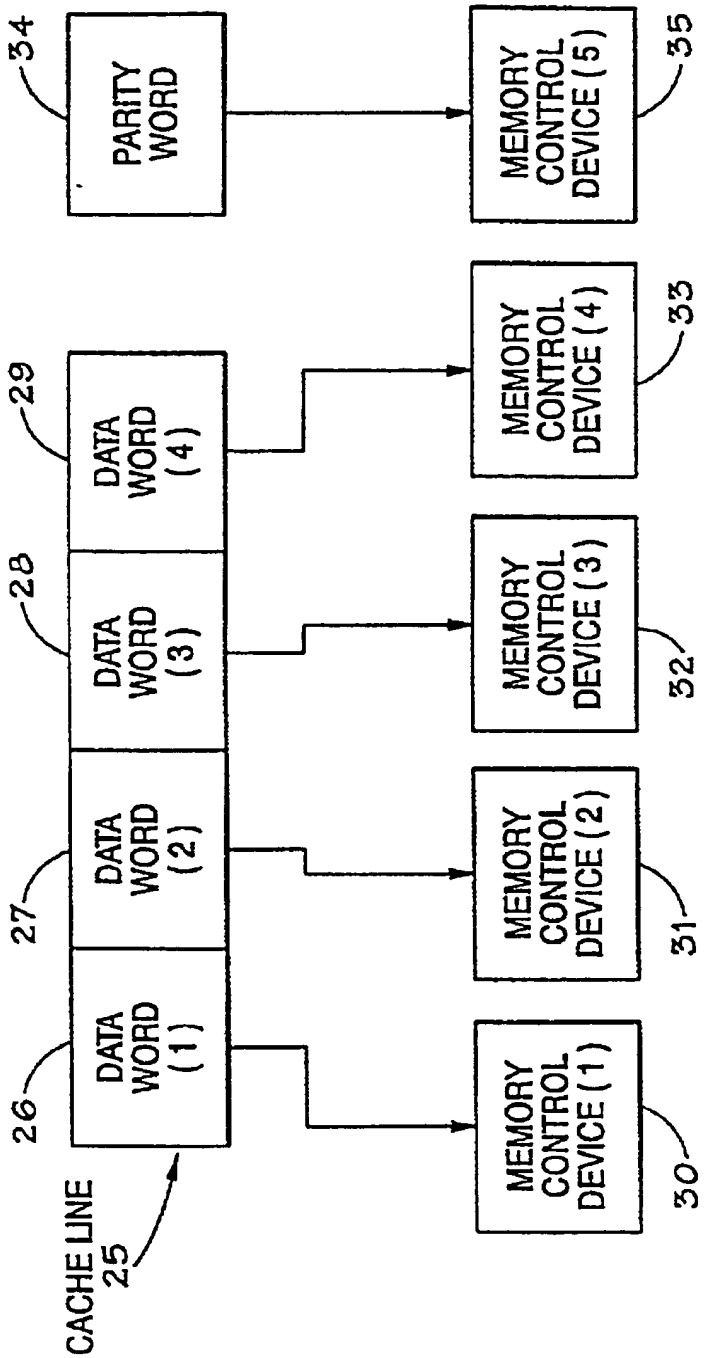

CACHE LINE
25
26
DATA WORD (1)
27
DATA WORD (2)
28
DATA WORD (3)
29
DATA WORD (4)
34
PARITY WORD
MEMORY CONTROL DEVICE (1)
30
MEMORY CONTROL DEVICE (2)
31
MEMORY CONTROL DEVICE (3)
32
MEMORY CONTROL DEVICE (4)
33
MEMORY CONTROL DEVICE (5)
35

FIG. 3

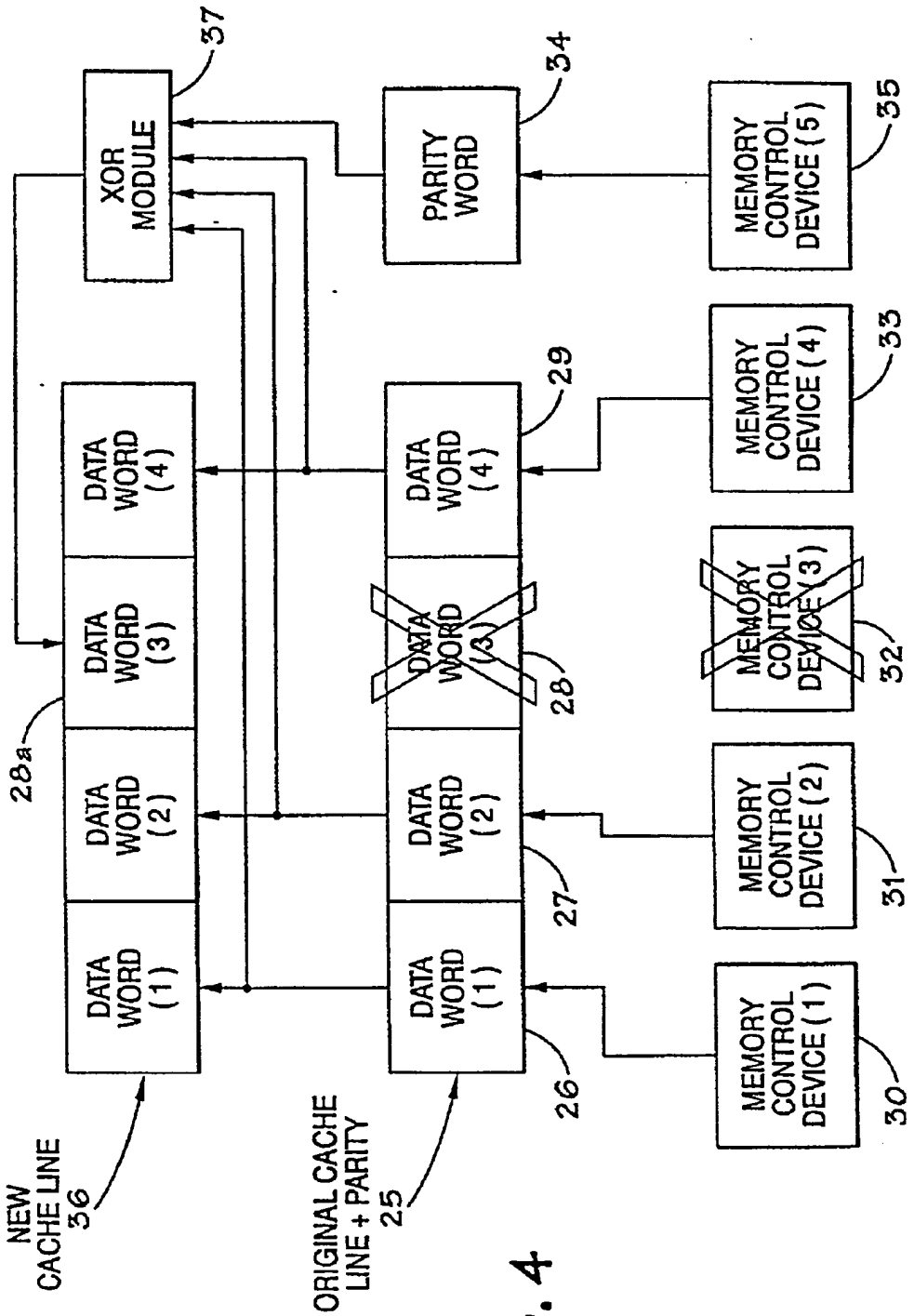

XOR MODULE
37
PARITY WORD
34
MEMORY CONTROL DEVICE (5)
35
DATA WORD (4)
DATA WORD (3)
DATA WORD (2)
DATA WORD (1)
28a
29
28
27
26
MEMORY CONTROL DEVICE (4)
33
MEMORY CONTROL DEVICE (3)
32
MEMORY CONTROL DEVICE (2)
31
MEMORY CONTROL DEVICE (1)
30
NEW CACHE LINE
36
ORIGINAL CACHE LINE + PARITY
25

FIG. 4

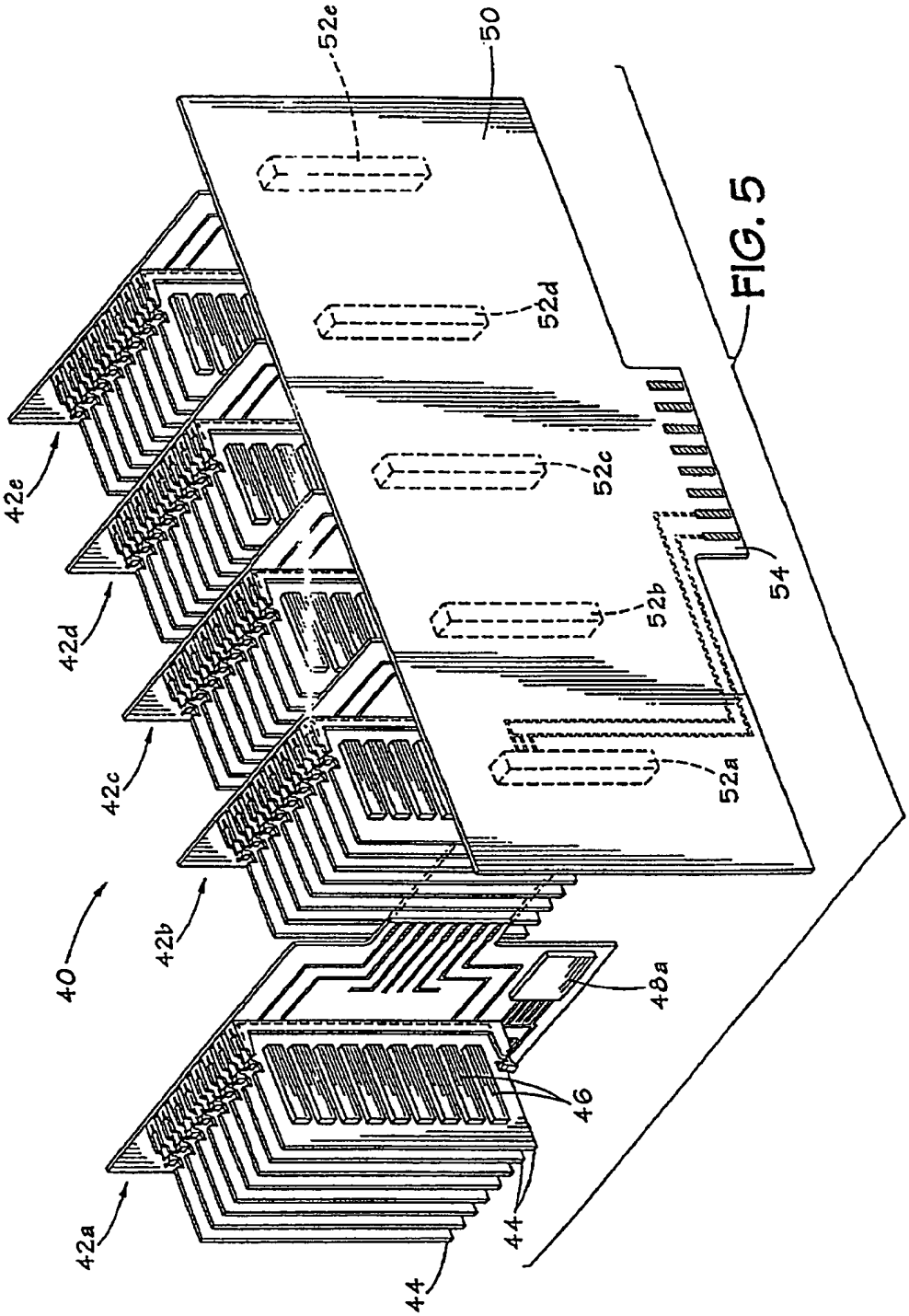


FIG. 5

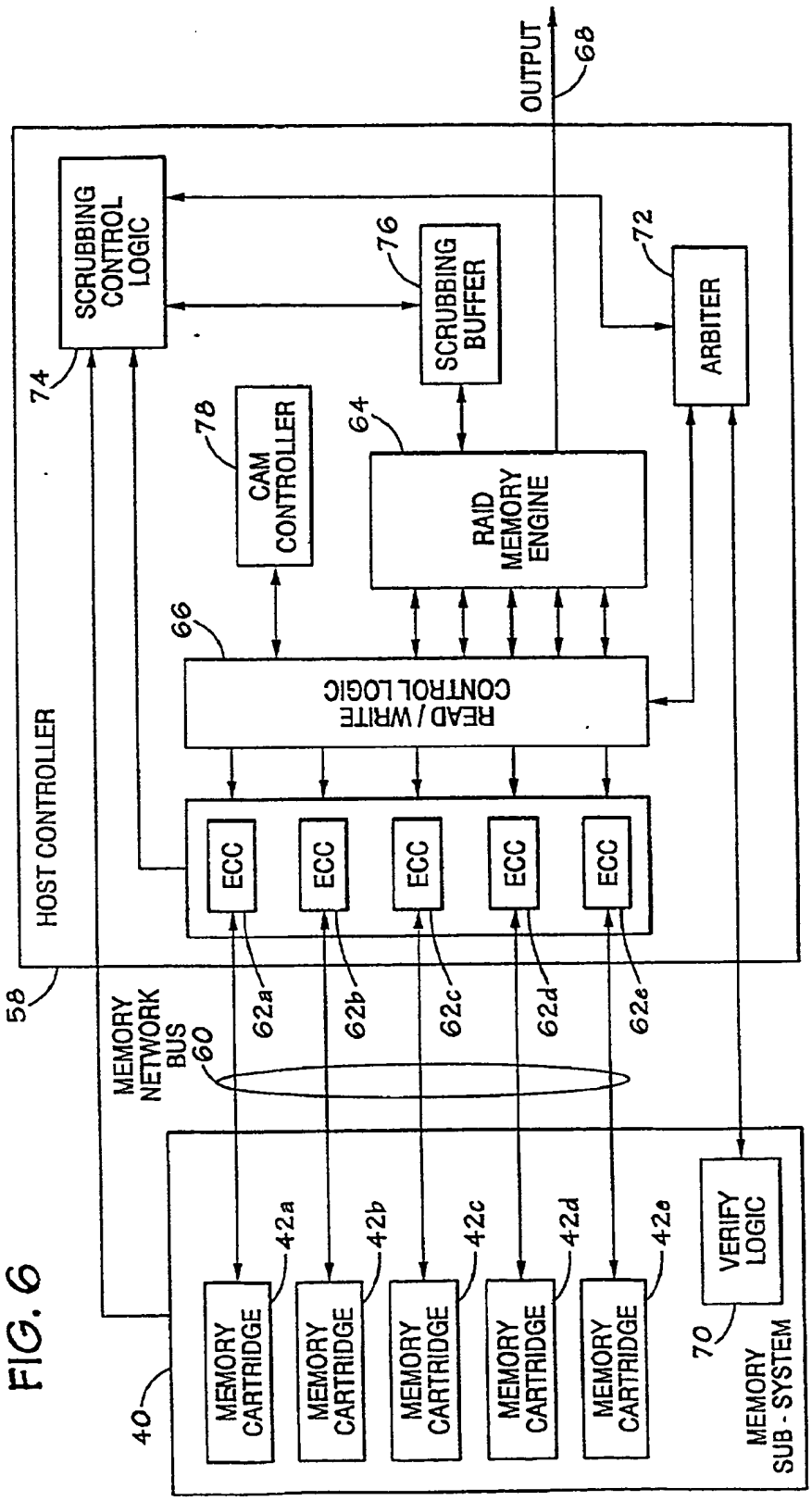

OUTPUT
68
SCRUBBING CONTROL LOGIC
74
SCRUBBING BUFFER
76
ARBITER
72
CAM CONTROLLER
78
RAID MEMORY ENGINE
64
66
READ / WRITE CONTROL LOGIC
ECC
ECC
ECC
ECC
ECC
HOST CONTROLLER
58
MEMORY NETWORK BUS
60
62a
62b
62c
62d
62e
MEMORY CARTRIDGE
42a
MEMORY CARTRIDGE
42b
MEMORY CARTRIDGE
42c
MEMORY CARTRIDGE
42d
MEMORY CARTRIDGE
42e
VERIFY LOGIC
70
MEMORY SUB - SYSTEM
40

FIG. 6

HARD/SOFT ERROR DETECTION

CROSS-REFERENCE TO RELATED APPLICATION

[0001] The present application is a continuation of application Ser. No. 09/769,958 filed on Jan. 25, 2001 which claims priority under 35 U.S.C §119(e) to provisional application 60/178,108 filed on Jan. 26, 2000.

BACKGROUND OF THE INVENTION

[0002] 1. Field of the Invention

[0003] The present invention relates generally to memory protection, and more specifically to a technique for detecting errors in a memory device.

[0004] 2. Description of the Related Art

[0005] This section is intended to introduce the reader to various aspects of art which may be related to various aspects of the present invention which are described and/or claimed below. This discussion is believed to be helpful in providing the reader with background information to facilitate a better understanding of the various aspects of the present invention. Accordingly, it should be understood that these statements are to be read in this light, and not as admissions of prior art.

[0006] Semiconductor memory devices used in computer systems, such as dynamic random access memory (DRAM) devices, generally comprise a large number of capacitors which store binary data in each memory device in the form of a charge. These capacitors are inherently susceptible to errors. As memory devices get smaller and smaller, the capacitors used to store the charges also become smaller thereby providing a greater potential for errors.

[0007] Memory errors are generally classified as "hard errors" or "soft errors." Hard errors are generally caused by issues such as poor solder joints, connector errors, and faulty capacitors in the memory device. Hard errors are reoccurring errors which generally require some type of hardware correction such as replacement of a connector or memory device. Soft errors, which cause the vast majority of errors in semiconductor memory, are transient events wherein extraneous charged particles cause a change in the charge stored in one of the capacitors in the memory device. When a charged particle, such as those present in cosmic rays, comes in contact with the memory circuit, the particle may change the charge of one or more memory cells, without actually damaging the device. Because these soft errors are transient events, generally caused by alpha particles or cosmic rays for example, the errors are not generally repeatable and are generally related to erroneous charge storage rather than hardware errors. For this reason, soft errors, if detected, may be corrected by rewriting the erroneous memory cell with correct data. Uncorrected soft errors will generally result in unnecessary system failures. Further, soft errors may be mistaken for more serious system errors and may lead to the unnecessary replacement of a memory device. By identifying soft errors in a memory device, the number of memory devices which are actually physically error free and are replaced due to mistaken error detection can be mitigated, and the errors may be easily corrected before any system failures occur.

[0008] Memory errors can be categorized as either single-bit or multi-bit errors. A single bit error refers to an error in a single memory cell. Single-bit errors can be detected and corrected by standard Error Code Correction (ECC) methods. However, in the case of multi-bit errors, which affect more than one bit, standard ECC methods may not be sufficient. In some instances, ECC methods may be able to detect multi-bit errors, but not correct them. In other instances, ECC methods may not even be sufficient to detect the error. Thus, multi-bit errors must be detected and corrected by a more complex means since a system failure will typically result if the multi-bit errors are not detected and corrected.

[0009] Regardless of the classification of memory error (hard/soft, single-bit/multi-bit), the current techniques for detecting the memory errors have several drawbacks. Typical error detection techniques rely on READ commands being issued by requesting devices, such as a peripheral disk drive. Once a READ command is issued to a memory sector, a copy of the data is read from the memory sector and tested for errors en route to delivery to the requesting device. Because the testing of the data in a memory sector only occurs if a READ command is issued to that sector, seldom accessed sectors may remain untested indefinitely. Harmless single-bit errors may align over time resulting in uncorrectable multi-bit errors. Once a READ request is finally issued to a seldom accessed sector, previously correctable errors may have evolved into uncorrectable errors thereby causing unnecessary data corruption or system failures. Early error detection may significantly reduce the occurrences of uncorrectable errors and prevent future system failures.

[0010] Further, in redundant memory systems, undetected memory errors may pose an additional threat. Certain operations, such as hot-plug events, may require that the system transition from a redundant to a non-redundant state. In a non-redundant state, memory errors which were of little concern during a redundant mode of operation, may become more significant since errors that were correctable during a redundant mode of operation may no longer be correctable while the system operates in a non-redundant state.

[0011] The present invention may address one or more of the concerns set forth above.

BRIEF DESCRIPTION OF THE DRAWINGS

[0012] The foregoing and other advantages of the invention will become apparent upon reading the following detailed description and upon reference to the drawings in which:

[0013] **FIG. 1** is a block diagram illustrating an exemplary computer system;

[0014] **FIG. 2** illustrates an exemplary memory device used in the present system;

[0015] **FIG. 3** generally illustrates a cache line and memory controller configuration in accordance with the present technique;

[0016] **FIG. 4** generally illustrates the implementation of a RAID memory system to recreate erroneous data words;

[0017] **FIG. 5** illustrates an exemplary memory sub-system in accordance with the present technique; and

[0018] FIG. 6 is a block diagram illustrating an exemplary architecture associated with a computer system in accordance with the present technique.

DETAILED DESCRIPTION OF SPECIFIC EMBODIMENTS

[0019] One or more specific embodiments of the present invention will be described below. In an effort to provide a concise description of these embodiments, not all features of an actual implementation are described in the specification. It should be appreciated that in the development of any such actual implementation, as in any engineering or design project, numerous implementation-specific decisions must be made to achieve the developers' specific goals, such as compliance with system-related and business-related constraints, which may vary from one implementation to another. Moreover, it should be appreciated that such a development effort might be complex and time consuming, but would nevertheless be a routine undertaking of design, fabrication, and manufacture for those of ordinary skill having the benefit of this disclosure.

[0020] Turning now to the drawings, and referring initially to FIG. 1, a multiprocessor computer system, for example a Proliant 8500 PCI-X from Compaq Computer Corporation, is illustrated and designated by the reference numeral 10. In this embodiment of the system 10, multiple processors 11 control many of the functions of the system 10. The processors 11 may be, for example, Pentium, Pentium Pro, Pentium II Xeon (Slot-2), or Pentium III processors available from Intel Corporation. However, it should be understood that the number and type of processors are not critical to the technique described herein and are merely being provided by way of example.

[0021] Typically, the processors 11 are coupled to a processor bus 12. As instructions are sent and received by the processors 11, the processor bus 12 transmits the instructions and data between the individual processors 11 and a host controller 13. The host controller 13 serves as an interface directing signals between the processors 11, cache accelerators 14, a memory controller 15 (which may be comprised of one or more memory control devices as discussed with reference to FIGS. 5 and 6), and an I/O controller 19. Generally, ASICs are located within the host controller 13. The host controller 13 may include address and data buffers, as well as arbitration and bus master control logic. The host controller 13 may also include miscellaneous logic, such as error detection and correction logic. Furthermore, the ASICs in the host controller may also contain logic specifying ordering rules, buffer allocation, specifying transaction type, and logic for receiving and delivering data. When the data is retrieved from the memory 16, the instructions are sent from the memory controller 15 via a memory bus 17. The memory controller 15 may comprise one or more suitable standard memory control devices or ASICs.

[0022] The memory 16 in the system 10 is generally divided into groups of bytes called cache lines. Bytes in a cache line may comprise several variable values. Cache lines in the memory 16 are moved to a cache for use by the processors 11 when the processors 11 request data stored in that particular cache line.

[0023] The host controller 13 is coupled to the memory controller 15 via a memory network bus 18. As mentioned above, the host controller 13 directs data to and from the processors 11 through the processor bus 12, to and from the memory controller 15 through the network memory bus 18, and to and from the cache accelerator 14. In addition, data may be sent to and from the I/O controller 19 for use by other systems or external devices. The I/O controller 19 may comprise a plurality of PCI-bridges, for example, and may include counters and timers as conventionally present in personal computer systems, an interrupt controller for both the memory network and I/O buses, and power management logic. Further, the I/O controller 19 is coupled to multiple I/O buses 20. Finally, each I/O bus 20 terminates at a series of slots or I/O interface 21.

[0024] Generally, a transaction is initiated by a requester, e.g., a peripheral device, via the I/O interface 21. The transaction is then sent to one of the I/O buses 20 depending on the peripheral device utilized and the location of the I/O interface 21. The transaction is then directed towards the I/O controller 19. Logic devices within the I/O controller 19 generally allocate a buffer where data returned from the memory 16 may be stored. Once the buffer is allocated, the transaction request is directed towards the processor 11 and then to the memory 16. Once the requested data is returned from the memory 16, the data is stored within a buffer in the I/O controller 19. The logic devices within the I/O controller 19 operate to read and deliver the data to the requesting peripheral device such as a tape drive, CD-ROM device or other storage device.

[0025] A system 10, such as a computer system, generally comprises a plurality of memory modules, such as Dual Inline Memory Modules (DIMMs). A standard DIMM may include a plurality of memory devices such as Dynamic Random Access Memory Devices (DRAMs). In an exemplary configuration, a DIMM may comprise nine memory devices on each side of the DIMM 22. FIG. 2 illustrates one side of a DIMM 22 which includes nine DRAMs 23. The second side of the DIMM 22 may be identical to the first side and may comprise nine additional DRAM devices (not shown). Each DIMM 22 access generally accesses all DRAMs 23 on the DIMM 22 to produce a data word. For example, a DIMM 22 comprising x4 DRAMs 23 (DRAMs passing 4-bits with each access) will produce 72-bit data words. System memory is generally accessed by CPUs and I/O devices as a cache line of data. A cache line generally comprises several 72-bit data words. Thus, each DIMM 22 accessed on a single memory bus provides a cache line of 72-bit data words 24.

[0026] Each of the 72 bits in each of the data words 24 is susceptible to soft errors. Different methods of error detection may be used for different memory architectures. The present method and architecture incorporates a Redundant Array of Industry Standard DIMs (RAID). As used herein, RAID memory refers to a "4+1 scheme" in which a parity word is created using an XOR module such that any one of the four data words can be re-created using the parity word if an error is detected in one of the data words. Similarly, if an error is detected in the parity word, the parity word can be re-created using the four data words. By using the present RAID memory architecture, not only can multi-bit errors be easily detected and corrected, but it also provides a system in which the memory module alone or the memory module and associated memory controller can be removed and/or

replaced while the system is running (i.e. the memory modules and controllers are hot-pluggable).

[0027] **FIG. 3** illustrates one implementation of RAID memory. RAID memory stripes a cache line of data **25** such that each of the four 72-bit data words **26, 27, 28,** and **29** is transmitted through a separate memory control device **30, 31, 32,** and **33**. A fifth parity data word **34** is generated from the original cache line **25**. Each parity word **34** is also transmitted through a separate memory control device **35**. The generation of the parity data word **34** from the original cache line **25** of data words **26, 27, 28,** and **29** can be illustrated by way of example. For simplicity, four-bit data words are illustrated. However, it should be understood that these principals are applicable to 72-bit data words, as in the present system, or any other useful word lengths. Consider the following four data words:

[0028] DATA WORD 1: 1 0 1 1

[0029] DATA WORD 2: 0 0 1 0

[0030] DATA WORD 3: 1 0 0 1

[0031] DATA WORD 4: 0 1 1 1

[0032] A parity word can be either even or odd. To create an even parity word, common bits are simply added together. If the sum of the common bits is odd, a "1" is placed in the common bit location of the parity word. Conversely, if the sum of the bits is even, a zero is placed in the common bit location of the parity word. In the present example, the bits may be summed as follows:

[0033] DATA WORD 1: 1 0 1 1

[0034] DATA WORD 2: 0 0 1 0

[0035] DATA WORD 3: 1 0 0 1

[0036] DATA WORD 4: 0 1 1 1

[0037] 2 1 3 3

[0038] Parity Word: 0 1 1 1

[0039] When summed with the four exemplary data words, the parity word 0111 will provide an even number of active bits (or "1's") in every common bit. This parity word can be used to recreate any of the data words (1-4) if a soft error is detected in one of the data words as further explained with reference to **FIG. 4**.

[0040] **FIG. 4** illustrates the re-creation of a data word in which a soft error has been detected in a RAID memory system. As in **FIG. 3**, the original cache line **25** comprises four data words **26, 27, 28,** and **29** and a parity word **34**. Further, the memory control device **30, 31, 32, 33,** and **35** corresponding to each data word and parity word are illustrated. In this example, a data error has been detected in the data word **28**. A new cache line **36** can be created using data words **26, 27,** and **29** along with the parity word **34** using an exclusive-OR (XOR) module **37**. By combining each data word **26, 27, 29** and the parity word **34** in the XOR module **37**, the data word **28** can be re-created. The new and correct cache line **34** thus comprises data words **26, 27,** and **29** copied directly from the original cache line **25** and data word **28*a*** (which is the re-created data word **28**) which is produced by the XOR module **37** using the error-free data words (**26, 27, 29**) and the parity word **34**. It should also be clear that the same process may be used to re-create a parity word **34** if an error is detected therein using the four error-free data words.

[0041] Similarly, if the memory control device **32**, which is associated with the data word **28**, is removed during operation (i.e. hot-plugging) the data word **28** can similarly be re-created. Thus, any single memory control device can be removed while the system is running or any single memory control device can return a bad data word and the data can be re-created from the other four memory words using an XOR module.

[0042] **FIG. 5** illustrates one embodiment of a memory sub-system **40**, which incorporates a redundant (4+1) scheme. The memory sub-system **40** comprises five memory cartridges **42*a-e***. Memory cartridge **42*e***, for example, may be used for parity storage. The memory cartridge **42*a*** includes eight DIMMs **44** mounted thereon. Each DIMM **44** includes nine memory devices, such as DRAMs **46** on each side of the DIMM substrate. (**FIG. 5** illustrates only one side of the DIMM **44**.) Further, the memory cartridge **42*a*** has a memory control device **48*a*** mounted thereon. It should be understood that each memory cartridge **42*a-e*** includes a plurality of DIMMs **44** and a corresponding memory control device **48**. The memory cartridges **42*a-e*** may be mounted on a memory system board **50** via connectors **52*a-e*** to create the memory sub-system **40**. The memory sub-system **40** can be incorporated into a computer system via an edge connector **54** or by any suitable means of providing a data path from the computer system to the memory storage devices **46**. It should be evident that each of the memory cartridges **42*a-e*** may be removed (hot-plugged) from the memory sub-system **40**. By removing a memory cartridge such as memory cartridge **42*a*** from the memory sub-system **40**, the computer system will transition from a redundant mode of operation (implementing the fifth memory cartridge) to a non-redundant state. When transitioning from a redundant to a non-redundant mode of operation during a hot-plug memory event, it may be advantageous to verify that no errors exist in the remaining memory cartridges **42*b-e***. Thus, immediately proceeding the removal of the memory cartridge **42*a***, a verify procedure may be advantageously implemented.

[0043] Further, a verify procedure may be advantageous in checking for memory errors in certain areas of memory which may sit idle for an extended period of time, allowing accumulation of errors or the growth of a single bit error to an uncorrectable multi-bit error. The verify procedure is implemented through a piece of logic which may reside in the memory sub-system **40**. The verify logic can be programmed to verify a specific region of memory such as the contents of a single memory cartridge **42*a-e*** or to verify the validity of the entire memory. The verify procedure relies on the normal ECC and error logging mechanisms to validate the health of the memory sub-system **40**. The verify routine may be exercised by an operator instruction, as part of a sequence of memory operations (such as a hot-plug event), or based on a predetermined schedule. Simply put, the verify logic will read a defined memory region. If errors are detected they may be recorded and corrected, as further discussed below with reference to **FIG. 6**. Verify may then be executed again to validate that the correction mechanism in fact corrected the errors that were reported. The verify

logic may reside in each memory controlled device **48***a-e* or on the memory system board **50**.

[0044] **FIG. 6** is a block diagram illustrating one embodiment of the verify technique which incorporates the RAID memory architecture. As previously described, a computer system includes a memory sub-system **40** comprising memory cartridges **42***a-e*. As described with reference to **FIG. 5**, each memory cartridge **42***a-e* may include a memory control device **48***a-e* (shown in **FIG. 5**). Thus, to access the memory devices **46** (shown in **FIG. 5**) in memory cartridge **42***a*, a READ command is issued and data is passed through the memory control device **48***a*, and so forth.

[0045] Each memory control device **48***a-e* may comprise ECC fault tolerance capability. As data is passed from the memory sub-system **40** to the host controller **58** via a memory network bus **60**, each data word being produced by a memory cartridge **42***a-e* is checked for single bit memory errors in each respective memory control device **48***a-e* (residing on each respective memory cartridge **42***a-e*) by typical ECC methods. If no errors are detected, the data is simply passed to the host controller **58** and eventually to a requesting device via an OUTPUT **68**. If a single-bit error is detected by a memory control device **48***a-e*, the data is corrected by the memory control device **48***a-e*. When the corrected data is sent to the host controller **58** via the memory network bus **60**, error detection and correction devices **62***a-e*, which reside in the first controller **58** and may be identical to the ECC devices in the memory control devices **48***a-e*, will not detect any erroneous data words since the single-bit errors have been corrected by the memory control devices **48***a-e* in the memory sub-system **40**. Therefore, if an error is detected and corrected by the memory control devices **48***a-e*, a message is sent from the memory control devices **48** are to the host controller **58** indicating that a memory error has been detected and corrected and that the corresponding memory cartridge **42***a-e* should be over-written with corrected data, as discussed in more detail below.

[0046] In an alternate embodiment, the error detection capabilities in the memory control devices **48***a-e* may be turned off or eliminated. Because the host controller **58** also includes error detection and correction devices **62***a-e*, any single bit errors can still be corrected using the standard ECC methods available in the host controller **58**. Further, it is possible that errors may be injected while the data is on the memory network bus **60**. In this instance, even if the error detection capabilities are turned on in the memory control devices **48***a-e*, the memory control devices **48***a-e* will not detect an error since the error is injected after the data has passed from the memory sub-system **40**. Advantageously, since the host controller **58** includes similar or even identical error detection and correction devices **62***a-e*, the errors can be detected and corrected in the host controller **58**.

[0047] If a multi-bit error is detected in one of the memory control devices **48***a-e*, the memory control device **48***a-e*, with standard ECC capabilities, can detect the errors but will not be able to correct the data error. Therefore, the erroneous data is passed to the error detection and correction devices **62***a-e*. Like the memory control devices **48***a-e*, the error detection and correction devices **62***a-e*, which also have typical ECC detection, can only detect but not correct the multi-bit errors. The erroneous data words may be passed to the RAID memory engine **64** via some READ/WRITE control logic **66**, for correction.

[0048] In a typical memory READ operation, the host controller **58** will issue a READ command on the memory network bus **60**, the READ command originating from an external device such as a disk drive. The memory control devices **48***a-e* receive the request and retrieve the data from the corresponding memory cartridge **42***a-e*. The data is then passed from the memory sub-system **40** to the host controller **58**. As described above, single-bit errors may either be corrected in the memory control devices **48***a-e* or the detection and correction devices **62***a-e*. The RAID memory engine **64** will correct the multi-bit errors, as described above. The corrected data will be delivered from the host controller **58** to the requesting controller or I/O device via an OUTPUT **68**.

[0049] It should be evident from the discussion above, that performing error detection and correction on data residing in the memory sub-system **40** by relying on READ operations sent from peripheral devices will only result in detection of errors on those devices from which data is read. By relying on the READ command from a peripheral device, certain areas of memory may sit idle for extended periods thereby allowing data errors to accumulate undetected. To address this issue, an additional piece of logic may reside in the memory sub-system **40**. The verify logic **70** initiates a routine based on an operator instruction, a pre-determined periodic instruction, or some sequence of events such as a hot-plug event, for example. The verify logic **70** initiates a check of the specified memory location in the memory sub-system **40** without depending on normal READ accesses by external devices.

[0050] The verify logic **70** initiates a verify procedure through an arbiter **72** in the host controller **58**. The arbiter **72** is generally responsible for prioritizing accesses to the memory sub-system **40**. A queue comprises a plurality of requests such as memory READ, memory WRITE, memory verify, and memory scrubs (discussed further below), for example. The arbiter **72** prioritizes the requests and otherwise manages the queue. The verify logic **70** essentially initiates its own internal READ command to check specified regions of the memory sub-system **40**. Once the verify logic **70** initiates a request to the arbiter **72**, the verify procedure is scheduled in the queue. The request will pass through the READ/WRITE control logic **66** and to the memory sub-system **40**. The specified memory locations in the memory sub-system **40** will be read and any errors will be detected and/or corrected by the means described above with reference to the READ command issued by a peripheral device. The verify procedure implemented by the verify logic **70** can be initiated in a variety of ways. For instance, a user may be able to check specified memory locations by pulling up a window on an operating system. The window may allow a user to specify what locations in memory the user would like checked. By providing a user with the ability to check specified memory locations, the verify procedure provides user confidence in the validity of data stored in the memory sub-system **40**.

[0051] Alternately, the verify procedure may be a periodically scheduled event. In this instance, the verify logic **70** may include a timer and a buffer for storing a list of each address location in the memory sub-system **40**. At pro-

grammed or specified time intervals, the verify logic **70** may initiate READ commands to the arbiter **72** to verify the data stored in the corresponding address locations in the memory sub-system **40**. The verify logic **70** may initiate READ commands through successive addresses in the memory sub-system **40** such that every memory address is eventually checked. The verify logic **70** thus may insure that all address locations in the memory sub-system **40** or a specified set of address locations are periodically checked for validity. Furthermore, the READ command issued by the verify logic **70** may be scheduled as a low priority thread in the arbiter **72** to minimize system impact. In this way, the verify procedure may only be run during periods of low system activity (e.g. when the queue in the arbiter **72** does not include READ/WRITE requests from external devices).

[0052] Yet another implementation of the verify logic **70** includes a verify operation to validate a memory cartridge when the memory sub-system **40** is switching from a non-redundant mode of operation to a redundant mode of operation (i.e. during a hot-plug event). For example, referring back to **FIG. 5**, the memory cartridges **42***b-e* are currently connected to the memory system board **50**. Assuming that the memory system board **50** is operably coupled to a host system including a host controller **58** (as illustrated in **FIG. 6**), the memory sub-system **40** is operating in a non-redundant mode since there is no additional memory cartridge **42***a* to be used for parity. If a memory cartridge **42***a* is installed into the memory sub-system **40**, it may be advantageous to verify the memory devices **46** residing on the memory cartridge **42***a*. The verify logic **70** can be implemented to check each address location on the memory devices **46** on the memory cartridge **42***a* before the system transitions to a redundant mode of operation.

[0053] First, the verify logic **70** initializes the memory cartridge **42***a* by writing zeros to each address location in the memory cartridge **42***a*. The verify logic **70** schedules the initialization WRITEs through the arbiter **70**. Next, the verify logic **70** rebuilds the memory cartridge **42***a* by using the techniques described in **FIGS. 3 and 4** to recreate the parity data that should be stored in the memory cartridge **42***a*. As previously described, each cache line of data from the memory cartridges **42***b-e* are used to recreate the parity cache line by using the XOR module in the RAID memory engine **64**. Each recreated cache line is then written to the corresponding location in the memory cartridge **42***a*. Finally, once the data in the memory cartridge **42***a* is rebuilt, the verify logic **70** may initiate a READ to insure that the data that should have been written to the memory cartridge **42***a* was in fact stored there. This procedure can be performed by again using the data stored in the memory cartridges **42***b-e* to again recreate the data that should be stored in the memory cartridge **42***a*, and then by comparing those values to the values that were stored in the memory cartridge **42***a* during the rebuild procedure. If the data does not match an error message may be provided to a user indicating that a DIMM on the memory cartridge **42***a* may be bad. If there are no errors found in the new memory cartridge **42***a*, the system may switch from a non-redundant mode of operation to a redundant mode of operation.

[0054] To this point, error detection via peripheral READ commands and READ commands implemented by the verify logic **70** have been discussed. The memory control devices **48***a-e*, the error detection and correction devices **62***a-e* and the RAID memory engine **64** can be used to correct the data before it is written to the output **68**. However, at this point the data residing in the memory sub-system **40** may still be corrupted. To rectify this problem, the data in the memory sub-system **40** may be overwritten or "scrubbed." For every data word in which a single bit error is detected and flagged by the memory control devices **48***a-e* or the error detection and correction devices **62***a-e*, a request is sent to the scrubbing control logic **74** indicating that the corresponding memory location should be scrubbed during a subsequent WRITE operation initiated by the scrubbing control logic **74**. Similarly, if a multi-bit error is detected by the error detection and correction devices **62***a-e*, the data is corrected through the RAID memory engine **64**, and the scrubbing control logic **74** is notified by the corresponding error detection and correction device **62***a-e* that the corresponding memory location in the memory sub-system **40** should be scrubbed. If a single-bit error is detected in one of the memory control devices **48***a-e*, or a multi-bit error is detected in one of the error detection and correction devices **62***a-e* a message is sent to the scrubbing control logic **74** indicating that an erroneous data word has been detected. At this time, the corrected data word and corresponding address location are sent from the RAID memory engine **64** to a buffer **76** which is associated with the scrubbing process. The buffer **76** is used to store the corrected data and corresponding address location temporarily until such time that the scrubbing process can be implemented. Once the scrubbing control logic **74** receives an indicator that a corrupted data word has been detected and should be corrected in the memory sub-system **40**, a request is sent to the arbiter **72** which schedules and facilitates all accesses to the memory sub-system **40**. To insure proper timing and data control, each time a data word is rewritten back to the memory sub-system **40**, an entire cache line may be rewritten into each of the corresponding memory cartridges **42***a-e* in the subsystem **40** rather than just rewriting the erroneous data word. The scrubbing logic can be used to rewrite the locations in the memory sub-system **40** when errors are found during a typical READ operation or a verify procedure initiated by the verify logic **70**.

[0055] Further, the host controller **58** may include a content addressable memory (CAM) controller **78**. The CAM controller **78** provides a means of insuring that memory WRITEs are only performed when necessary. Because many READ and WRITE requests are active at any given time on the memory network bus **60** and because a scrubbing operation to correct corrupted data may be scheduled after a WRITE to the same memory location, the CAM controller **78** will compare all outstanding WRITE requests to subsequent memory scrub requests which are currently scheduled in the queue. It is possible that a corrupted memory location in the memory sub-system **40** which has a data scrub request waiting in the queue may be overwritten with new data prior to the scrubbing operation to correct the old data previously present in the memory sub-system **40**. In this case, the CAM controller **78** will recognize that new data has been written to the address location in the memory sub-system **40** by implementing a simple compare function between the addresses and will cancel the scheduled scrubbing operation. The CAM controller **78** will insure that the old corrected data does not over-write new data which has been stored in the corresponding address location in the memory sub-system **40**.

[0056] It should be noted that the error detection and scrubbing technique described herein may not distinguish between soft and hard errors. While corrected data may still be distributed through the output of the host controller **58**, if the errors are hard errors, the scrubbing operation to correct the erroneous data words in the memory sub-system **40** will be unsuccessful. To solve this problem, software in the host controller **58** may track the number of data errors associated with a particular data word or memory location. After some pre-determined number of repeated errors are detected in the same data word or memory location, the host controller **58** may send an error message to a user or illuminate an LED corresponding to the device in which the repeat error is detected.

[0057] While the invention may be susceptible to various modifications and alternative forms, specific embodiments have been shown by way of example in the drawings and will be described in detail herein. However, it should be understood that the invention is not intended to be limited to the particular forms disclosed. Rather, the invention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the invention as defined by the following appended claims.

What is claimed is:

1. A method of tracking errors in a memory system comprising the acts of:

detecting an error in a semiconductor memory segment;

determining an error type, the error type being one of a soft error and a hard error;

counting the number of hard errors detected in the memory segment; and

indicating that a threshold number of hard errors has been reached.

2. The method of tracking errors, as set forth in claim 1, wherein the act of detecting an error comprises the act of detecting an error using an ECC algorithm.

3. The method of tracking errors, as set forth in claim 1, wherein the act of detecting an error comprises the act of detecting an error in a dual inline memory module (DIMM).

4. The method of tracking errors, as set forth in claim 1, wherein the act of detecting an error comprises the act of detecting an error during execution of a READ request.

5. The method of tracking errors, as set forth in claim 1, wherein the act of detecting an error type comprises the acts of:

writing corrected data to a memory segment address corresponding to the error;

reading the corrected data from the memory segment address corresponding to the error; and

performing error detection on the corrected data read from the memory segment address corresponding to the error, wherein if a second error is detected, defining the error as a hard error.

6. The method of tracking errors, as set forth in claim 1, wherein the act of counting comprises the act of incrementing a counter each time a hard error is detected.

7. The method of tracking errors, as set forth in claim 1, wherein the act of indicating comprises the act of illuminating a light emitting diode (LED).

8. The method of tracking errors, as set forth in claim 1, wherein the act of indicating comprises the act of indicating that the threshold number of hard errors has been reached, the threshold number corresponding to an indication that the memory segment having the hard errors should be replaced.

9. The method of tracking errors, as set forth in claim 1, comprising the act of selecting the threshold, wherein the threshold number corresponds to a user-selectable maximum number of hard errors corresponding to an indication that the memory segment having the hard errors should be replaced.

10. An error detection system comprising:

a plurality of semiconductor memory segments;

a plurality of memory controllers, wherein each of the memory controllers is operably coupled to a corresponding one of the plurality of memory segments and configured to initiate requests to the respective memory segment;

error detection logic configured to detect errors during execution of a first READ request, wherein the errors comprise one of a soft error and a hard error in the plurality of memory segments; and

a counting device configured to count only when a hard error is detected.

11. The error detection system, as set forth in claim 10, wherein each of the plurality of memory segments comprises a dual inline memory module (DIMM).

12. The error detection system, as set forth in claim 11, comprising a light emitting diode (LED) corresponding to each of the dual inline memory modules (DIMMs), wherein each of the LEDs is configured to illuminate in response to the counting device reaching a threshold number N of hard errors for the respective DIMM.

13. The error detection system, as set forth in claim 12, wherein the threshold number N is user-programmable.

14. The error detection system, as set forth in claim 10, wherein each of the plurality of memory controllers is configured to initiate a WRITE request in response to an error being detected, the WRITE request being initiated to write corrected data to an address corresponding to the detected error.

15. The error detection system, as set forth in claim 14, wherein each of the plurality of memory controllers is configured to initiate a second READ request after the WRITE request, the READ request being initiated to read the corrected data.

16. The error detection system, as set forth in claim 15, wherein the error detection logic is configured to detect errors during execution of the second READ request.

17. The error detection system, as set forth in claim 16, wherein the counting device is configured to count when an error is detected in the data corresponding to the second READ request.

18. The error detection system, as set forth in claim 10, wherein each of the plurality of memory controllers comprises the error detection logic.

19. A method of manufacturing a memory system, comprising the acts of:

providing a device to detect hard errors in the memory system;

providing a device to count the number of hard errors detected in the memory system; and

providing an indication device to indicate that a threshold number of hard errors have been counted.

20. The method of manufacturing, as set forth in claim 19, comprising the acts of:

providing a plurality of memory segments; and

providing a plurality of memory controllers, each of the plurality of memory controllers corresponding to one of the plurality of memory segments, and wherein each of the plurality of memory controllers is configured to provide access to the memory segments.

21. The method of manufacturing, as set forth in claim 19, wherein the act of providing a device to detect hard errors comprises the act of providing a device comprising an FCC algorithm.

22. The method of manufacturing, as set forth in claim 19, wherein the act of providing a device to detect hard errors comprises the act of providing a memory controller comprising and ECC algorithm.

23. The method of manufacturing, as set forth in claim 20, wherein the act of providing a device to count the number of hard errors detected comprises the act of providing a counter configured to increment by one each time a hard error is detected.

24. The method of manufacturing, as set forth in claim 20, comprising providing a configuration register configured to store the threshold number of errors, the threshold number of errors corresponding to a maximum number of errors that may be detected without indicating a memory segment error.

25. The method of manufacturing, as set forth in claim 20, wherein the act of providing an indication device to indicate that a threshold number of hard errors comprises the act of providing a light emitting diode (LED) corresponding to each of the plurality of memory segments and configured to illuminate when the threshold number of errors has been detected.

* * * * *